Python routines for low temperature resistive thermometry in magnetic fields

Nathanael A. Fortune[1], Scott Hannahs[2], and Julia Frothingham[3]

[1]Smith College
[2]Florida State University
[3]Affiliation not available

July 16, 2021

1 Introduction

1.0.1

1.1 Thermometer calibration

Following the method outlined in Ref. (Fortune et al., 2000), we approximate the magnetic field and temperature dependence $R\left(T,\ B\right)$ of a resistive thermometer using a linear combination of Chebyshev polynomials $t_n\left(x\right)$

$$\ln R(T,B) \simeq \sum_{i=0}^{N} c_i(B) t_i(x) \tag{1}$$

where x is a function of $\ln\left(T\right)$ and the polynomial coefficients $c_n\left(B\right)$ are a function of magnetic field B. Since Chebyshev polynomials are defined over the window $-1\ \leq x\ \leq 1$, we define x as

$$x = \frac{\left(\ln T - \ln T_{\min}\right) - \left(\ln T_{\max} - \ln T\right)}{\left(\ln T_{\max} - \ln T_{\min}\right)} \tag{2}$$

where $T_{\min}$ and $T_{\max}$ correspond to the minimum and maximum temperatures over which the fit will be defined.

To account for the magnetic field dependence, we allow for the fitting coefficients c_i in the Chebyshev expansion to vary in value with magnetic field; we calculate the magnetic field dependence of each Chebyshev coefficient $c_i\left(B\right)$ by fitting $R\left(T\right)$ data at a series of fixed magnetic fields and expressing the resulting $c_i\left(B\right)$ coefficients as rational polynomials (Padé approximants) . Specifically, we represent the fractional variation of $c_i\left(B\right)$ with respect to its zero field value $c_i\left(0\right)$ as

$$y_i = \frac{c_i(B) - c_i(0)}{c_i(0)} = \frac{\sum_{p=0}^{P} \kappa_{i,p} B^p}{\sum_{q=0}^{Q} \gamma_{i,q} B^q} \tag{3}$$

with $\kappa_{i,0} \equiv 0$ and $\gamma_{i,0} \equiv 1$ for all values of i .

Solving for $c_i\,(B)$,

Rewriting Eq. 1 in terms of Eq. 3,

$$c_i(B) = c_i(0)\,[1 + y_i(B)] = c_i(0)\left[1 + \frac{\sum_{p=0}^{P} \kappa_{i,p} B^p}{\sum_{q=0}^{Q} \gamma_{i,q} B^q}\right] \tag{4}$$

and rewriting Eq. 1 in terms of Eq. 4,

$$\ln R(T, B) \simeq \sum_{i=0}^{N} [1 + y_i(B)]\, c_i(0) t_i(x) \tag{5}$$

The success of the fit ultimately depends on proper choice of P, Q, and initial values for $\kappa_{i,p}$ and $\gamma_{i,q}$. Guidance on how to choose these values is provided in section 4 : R(B,T) calibration.

1.2 Thermometer Sensitivity

Since the first derivative of a Chebyshev polynomial is also a Chebyshev polynomial (Karageorghis, 1988; Fortune et al., 2000; Barrio, 2004), both R and the dimensionless sensitivity η can be directly calculated from T once the values of c_i have been determined. Specifically,

$$\eta = \frac{d \ln R}{d \ln T} = \left(\frac{2}{\ln T_{max} - \ln T_{min}}\right) \sum_{j=0}^{N-1} d_n(B) t_n(x) \tag{6}$$

where the coefficients d_n can be conveniently re-expressed in terms of the original coefficients c_n as

$$d_n = a_n \sum_{j=1}^{N-1} (n + 2j - 1)\, c_{n+2j-1} \tag{7}$$

with $a_0 = 2$ and $a_n = 4$ for $n \geq 1$ (Karageorghis, 1988). Conveniently, computation of R and η can be done for us automatically from the Chebyshev coefficients c_n for any given T in the domain $[T_{\min},\ T_{\max}]$ using the Python routines presented below.

2 Zero-field R(T) calibration

In this example, we calculate Chebyshev coefficients from zero field calibration data for a commercially calibrated resistive thermometer. In later examples, we determine the corresponding coefficient values in non-zero magnetic fields. To see and run the underlying Python code, click on the </> Code button to the left of Fig. 1 below.

2.1 Load Python packages for data fitting

For these fits, we use the Chebyshev convenience class routines included in the Numerical Python (Numpy) Polynomial package. Please see the Numerical Python reference manual and the example code attached to Fig.1 below for further details.

```
# import numerical Python package
import numpy as np

# import Chebyshev polynomial routines
from numpy.polynomial import Chebyshev

# import nonlinear curve fitting routine from SciPy package
from scipy.optimize import curve_fit
```

2.2 Import the data

The commands below indicate that the data is in a tab delimited two-column spreadsheet format consisting of consecutive pairs of (T, R) data pairs (preceded by 10 rows of header text). For additional details — including how to import other forms of spreadsheet data such as comma separated variable (CSV) files — please read *A Short Guide to Using Python for Data Analysis in Experimental Physics*.

```
T_CT_cal_data, R_CT_cal_data  = np.loadtxt('Smith Cernox 1010 X65735LF.txt', skiprows =10, unpack = Tru
```

To see the original calibration data file (including header information), click on the Data icon to the left of Fig.1 below.

Since orthogonal polynomials such as the Chebyshev polynomials are defined over a certain domain, we first need to establish domain and range of the data.

```
T_CT_limits = np.array([np.min(T_CT_cal_data), np.max(T_CT_cal_data)])
R_CT_limits = np.array([np.min(R_CT_cal_data), np.max(R_CT_cal_data)])

print('temperature range of calibration data [K]', T_CT_limits)
print('resistance range of calibration data [ohms]', R_CT_limits)
```

With results

```
temperature range of calibration data [K] [8.45954187e-02 3.30011773e+02]
resistance range of calibration data [ohms] [3.33321191e+01 2.01328000e+05]
```

2.3 Chebyshev polynomial fit

All of the data needs to fit within the domain of temperatures we choose for the Chebyshev fit. For convenience, we extrapolate slightly on both the low and high temperature end to the range of temperatures we expect to regularly encounter. Too large an extrapolation will lead to poor fits to the data and/or implausible extrapolations, so the setting of the domain can involve some trial and error.

```
def set_domain(T_min, T_max):
    """set range of temperatures for which fit is to be done"""
    domain = np.array([T_min,T_max])
    return domain

CT_domain = set_domain(0.050, 335) # defines lower and upper bounds of fit, in Kelvin
```

In the example below, we use the numpy Chebyshev polynomial class method `Chebyshev.fit` to fit an N = 8 degree (9 term) Chebyshev polynomial to the calibration data (in the manner of Eq. 1). For further details on `Chebyshev.fit` method, see the Numpy reference manual entry.

```
def fit_Chebyshev_to_data(T_data, R_data, N, T_domain):
    """find coefficients for an Nth order logR = f(x(logT)) Chebyshev fit """
    fit = Chebyshev.fit(np.log(T_data),np.log(R_data), N, domain = np.log(T_domain))
    return fit

CT_fit = fit_Chebyshev_to_data(T_CT_cal_data, R_CT_cal_data, 8, CT_domain) # compare to manual fit
```

Notice that the scaling of the temperature domain $[\log T_{\min}, \log T_{\max}]$ to the window [-1, 1] for x — as described in Eq. 2 — is handled automatically by `Chebyshev.fit`. We do not need to write a function to explicitly calculate x from T.

The result of the fit is shown in Fig. 1 below. A Jupyter notebook containing the Python code used to generate this and following figures is attached to Fig.1. To open it, click on the </> Code button to the left of the figure to open a Jupyter notebook session, then click on `R(T)_fit_part_1.ipynb` to launch the

notebook. The notebook can be run as is (for example, by selecting **Restart and Run All** from the **Kernel** menu) or modified for use with your own data.

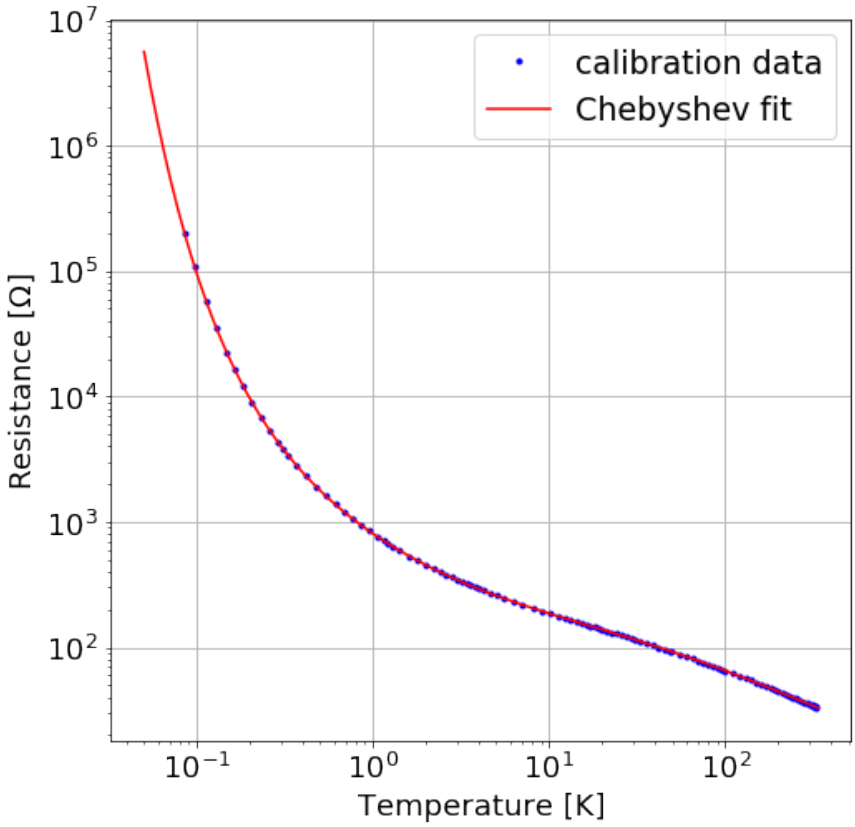

Figure 1: Original calibration data and 8th degree Chebyshev polynomial fit. In this particular case the operable temperature domain for the fit was set to [50 mK, 335 K], resulting in a slight extrapolation of the data at the low temperature end.

The corresponding calculated dimensionless sensitivity is shown in Fig. 2.

2.4 Evaluation and Assessment of fit

2.4.1 numerical evaluation of R(T)

We can evaluate this fit for any temperature value `T_value`, using `CT_fit(np.log(T_value))`.

```
T_value = 1.0 # Kelvin
R_value = np.exp(CT_fit(np.log(T_value)))
print(R_value) # in ohms
```

```
804.8943283874187
```

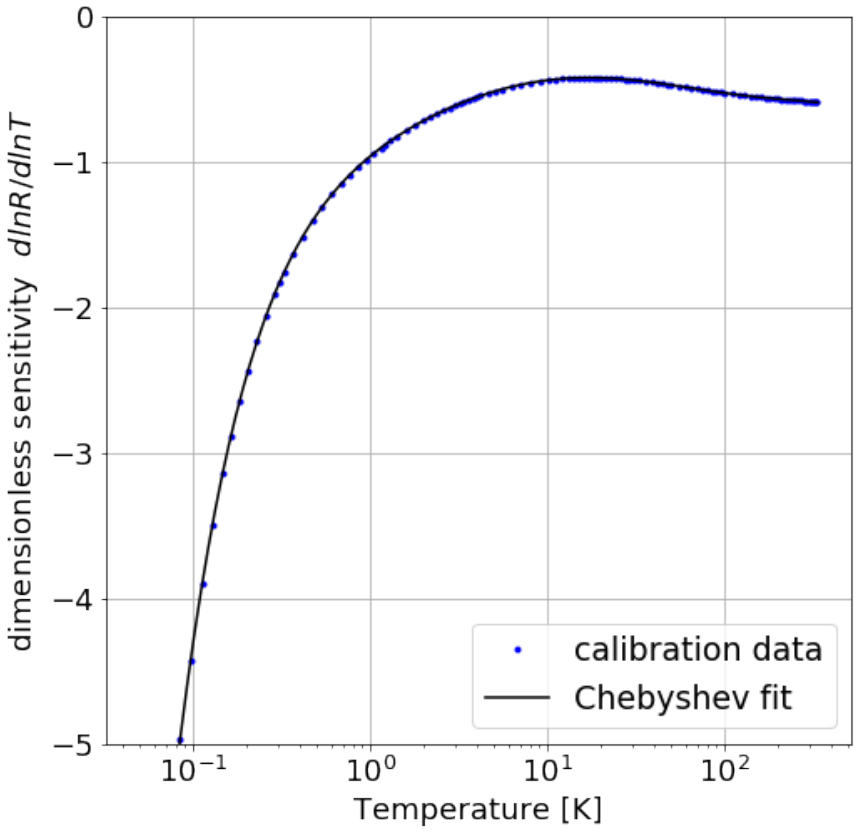

Figure 2: Variation of the dimensionless sensitivity $\eta(T) = \frac{d \ln R}{d \ln T}$ with temperature T, as described in Eq. 6. The values are calculated from the Chebyshev fit to Eq.1 for the original $R(T)$ calibration data using the differentiation function built into the Chebyshev.fit method.

This one line command first calculates $\ln T$ from T, then evaluates the Chebyshev fit to find $\ln R$, and finally calculates R. To calculate the resistances corresponding to an array of temperature values all at once, we simply replace the variable 'T_value' with a numpy array of temperature values.

For convenience, we define a function `TtoR` that will convert an array of temperature values to an array of resistance values for any given set of Chebyshev coefficients using the steps outlined above:

```
def TtoR(T_values, Chebyshev_fit):
    """use fit to calculate R from T"""
    lnR = Chebyshev_fit(np.log(T_values))
    R_values = np.exp(lnR)
    return R_values
```

2.4.2 fractional error in resistance

We can use this function to find the fractional error in R for the original calibration data resistance values:

```
R_calc = TtoR(T_CT_cal_data, CT_fit)
fractional_error_in_R = (R_calc - R_CT_cal_data)/ R_CT_cal_data
```

The result, shown in Fig. 3, indicates that the fractional error in the fit is less than ± 0.2% over most of the temperature range, but starts to break down at the low temperature end. To formally evaluate the goodness of fit, we would need to know the precision of the resistance measurements for each data point. The precision of the data points can be taken into account during the Chebyshev fitting process by specifying the appropriate weights in `Chebyshev.fit`.

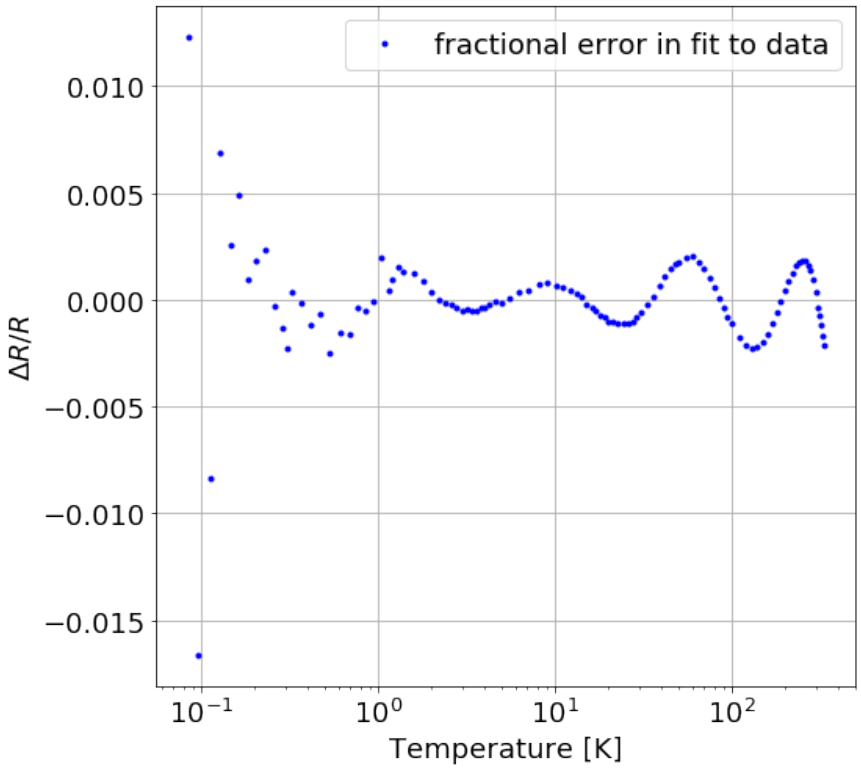

Figure 3: Comparison of the resistance values calculated using the Chebyshev fit with the measured resistance values from the calibration data, expressed as the fractional resistance $\frac{\Delta R}{R} = \frac{R_{calc} - R_{data}}{R_{data}}$. The fractional error is less than ±0.2% over most of the temperature range.

2.4.3 numerical evaluation of T(R)

The ultimate purpose of the $R\,(T)$ calibration is to be able to determine the temperature from a measurement of the thermometer resistance. Rather than carry out a second Chebyshev fit for $T\,(R)$ — which can lead to inconsistencies in the handling of the data — we instead find the value of T for which our calibration calculates a value of R corresponding to the measured R value by use of the `Chebyshev.roots()` root finding routine built into `Chebyshev_fit`.

Suppose then that for $x^* = x\,(T^*)$, we find that $R_{calc}\,(T^*) = R_{measured}\,(T_{measured})$. Then

$$\ln R(T,B) = \sum_{i=0}^{N} c_i(B) t_i(x^\star) \tag{8}$$

and since $t_0(x) = 1$ for all x within the window [-1, 1],

$$(\ln R_{\text{measured}} - c_0) - \sum_{i=1}^{N} c_i(B) t_i(x^\star) = \sum_{i=0}^{N} c_i^\star(B) t_i(x^\star) = 0 \tag{9}$$

where $c_0^* = \ c_0 \ - \ln R$ and $c_i^* = c_i$ for $i \ \geq 1$. The roots x^* are the values for which Eq. 9 is satisfied.

The key steps in the Numpy Chebyshev_fit root finding routine for the evaluation of $T(R)$ are these:

1. make a copy of the Chebyshev results so as not to modify the original
2. replace c_0 with c_0^* so that 0 is returned when series is evaluated for correct value of T
3. find all roots, then isolate the (ideally) one real root within the temperature domain of the fit

The key parts of the code corresponding to each of these step are listed here:

```
#make a copy so as not to modify the original
root_fit = Chebyshev_fit.copy()

# adjust coefficients so that zero is returned when series is evaluated
# if T is correct value
root_fit.coef[0] = Chebyshev_fit.coef[0] - np.log(value)

# find all roots, real and complex
roots = root_fit.roots()  # compute roots for a particular R value

# isolate the one real root within T domain (assuming root finding is successful)
  T_solve = [np.real(np.exp(root)) for root in roots
  if ((root < Chebyshev_fit.domain.max()
  and root > Chebyshev_fit.domain.min())
  and np.isreal(root))]
```

These commands are included in the complete code for the root finding function `RtoT` listed here:

```
def RtoT(R_values, Chebyshev_fit):
    """use fit to calculate T from R for set of R_values
    by finding roots of Chebyshev series for modified coefficient
    root_fit.coef[0]= Chebyshev_fit.coef[0] - np.log(R_value)
    then return as a set of T_values"""

    # fix if user inputs a scalar value instead of an array for R_values
    if np.isscalar(R_values):
        values = np.array([R_values])
    else:
```

```
        values = R_values

    # prepare to find roots
    root_fit = Chebyshev_fit.copy()  #make a copy so as not to modify the original
    T_values = np.empty(values.size) #create an empty array to fill with T_values

    for index, value in enumerate(values):
        root_fit.coef[0] = Chebyshev_fit.coef[0] - np.log(value)
        roots = root_fit.roots()  # compute roots for R value

        #sort roots, find root that is real and within T domain
        T_solve = [np.real(np.exp(root)) for root in roots if ((root < Chebyshev_fit.domain.max() and r

        #check that there is one and only one real root
        if len(T_solve) == 1:
            T_value = T_solve[0]
        elif len(T_solve) == 0:
            print('error! no real roots for R = ', value)
            T_value = np.nan
        else:
            print('warning! multiple real roots in domain for R =', value)
            print(T_solve)
            T_value = np.nan
        # assign T_value to array
        T_values[index]   = T_value   # assign T value
    return T_values
```

2.4.4 fractional error in temperature

We can now calculate the fractional and percentage errors in temperature for each of the original calibration (R,T) data pairs.

```
T_calc = RtoT(R_CT_cal_data, CT_fit)
fractional_error_in_T = (T_calc - T_CT_cal_data)/ T_CT_cal_data
percent_error_in_T = fractional_error_in_T * 100
```

A graph of the *percentage error* for each of the calibration data points for this particular Chebyshev fit is shown in Fig. 4 below. Notice that although the fractional error in resistance is largest at the lowest temperatures, the fractional error in temperature is actually largest at high temperature (because of the lower sensitivity). The Jupyter notebook containing the Python code used to calculate T from R and to generate this figure is attached to Fig. 1 above.

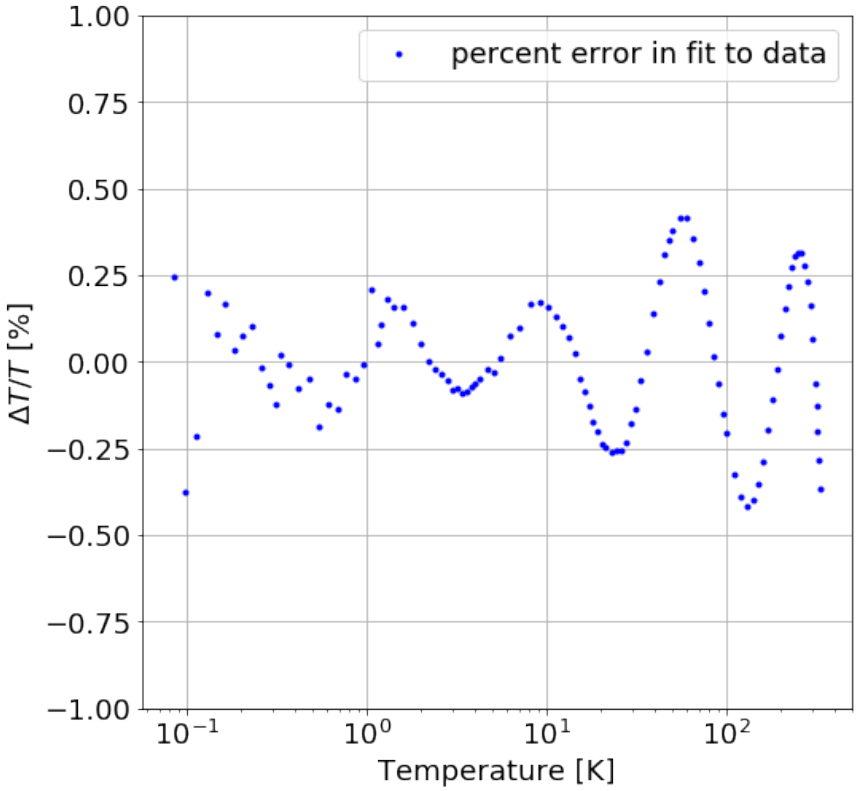

Figure 4: Fractional error in temperature $\frac{\Delta T}{T} = \frac{T_{calc} - T_{data}}{T_{data}}$ for our fit to the data in Fig.1, expressed as a percentage.

3 T(P) calibration

3.1 Introduction

One method of calibrating a resistive thermometer in magnetic field over the range $[T_{\min}, T_{\max}]$ is to thermally sink the thermometer to a temperature controlled platform which is in turn weakly linked to a constant temperature cryogenic bath at T_0. If the temperature of the platform and the base temperature of the bath are independent of magnetic field strength, then we expect the temperature of the platform to be a magnetic-field-independent function of platform heater power P_{heater}.

Experimentally, this assumption can be expected to break down

1. if fluctuations in T_0 cause significant changes in T_{platform} at constant P_{heater}
2. if the thermal conductance of the 'weak' thermal link is too large or rises too quickly with temperature
3. if ramping of the field changes the temperature of the platform during measurements

Problem 1 is most likely to occur at low T, as $T_{\min} \to T_0$, so we require $T_{\min} \gg T_0$.

Problem 2 is most likely to occur at high T, so we require that P_{heater} not exceed the cooling power of the cryogenic bath at the maximum desired value of T_0.

Problem 3 arises from a changing magnetic field, so we require that the magnetic field remain constant as $P_{\text{platform heater}}$ (and hence T) are varied and that the platform and cryogenic system be allowed to return to equilibrium after a change in field prior to ramping the heater power.

These considerations lead to use of the "bootstrap" method of calibrating resistive thermometers in field, in which $R(P, B)$ is measured as a function of platform heater power P while the magnetic field B is held constant, then repeated for a series of different magnetic field values. The method acquires its name because we first create a functional fit for $T(P)$ in zero field (using a previously calibrated thermometer), then use that $T(P)$ calibration to generate $R(T, B)$ from R(P,B)$R(P, B)$, in a method analogous to booting up a computer by loading the operating system software into the memory, then using that software to take care of loading other software as needed.

We proceed by using Python to prepare a calibration of $P\,(T)$ from measurements of $R\,(P)$ and our zero field $R(T)$ Chebyshev fit.

3.2 Python code

3.2.1 calculate T from R

Import zero field $R\,(P)$ data for the calibrated thermometer

```
P_PH, R_CT = np.loadtxt('CT_0_kg.txt', skiprows =1, delimiter=',', unpack = True)
```

Import the zero field $R\,(T)$ calibration for the thermometer

```
calibration_file_folder = ''  # if file in same folder as program, else
# calibration_file_folder = 'calibrations/'

calibration_file_name = 'CT_CX1010_SN_X65735LF_50mK_335K'
pickled_file = calibration_file_folder + calibration_file_name + '.p'

with open(pickled_file, "rb") as f:
    CT_fit = pickle.load(f)
```

use zero calibration to find temperature

```
T_CT = RtoT(R_CT, CT_fit)
```

3.2.2 establish limits for $T\,(P)$ fit

We first set the domain over which our $T\,(P)$ fit will apply. Querying the data,

"' P_CT_limits = np.array([np.min(P_PH), np.max(P_PH)])

print(‘range of platform heater powers: [’ , P_CT_limits[0],’,’, P_CT_limits[1], ’] Watts’)

we find for this data set that

```
range of platform heater powers: [ 1.4e-08 , 0.005 ] Watts
```

and as usually choose to set the domain for fits over a slightly larger region:

```
PH_domain = np.array([5E-9, 1E-2])
```

3.2.3 carry out Chebyshev fit

By trial and error, we find that an N = 7 degree (8th order) fit is the highest order fit for which the coefficients converge to ever smaller values (after the first term).

```
PH_fit = Chebyshev.fit(np.log(P_PH),np.log(T_CT), 7, np.log(PH_domain))

print(PH_fit.coef)
```

with results

```
[-2.27672039e-01  2.52795517e+00  1.96265839e-01  9.25867825e-02
  4.77725069e-02  9.81698921e-03  8.25879812e-03 -1.25018420e-03]
```

We now generate an array of values using the fit and compare then to the original data:

```
PH_lnP, PH_lnT = PH_fit.linspace(n=100)
P_PH_fit = np.exp(PH_lnP)
T_PH_fit = np.exp(PH_lnT)
```

The results are shown in Fig. 5 .

3.2.4 assess results of $T(P)$ fit

To assess the results, we calculate the fractional error in temperature.

```
T_PH = np.exp(PH_fit(np.log(P_PH)))
T_PH_fractional_error = (T_PH - T_CT)/T_CT
```

In this case, the two temperature calculations agree within 0.2 % over the temperature range 0.1 to 10 K.

3.2.5 save result

```
# CT calibration using LS zero field data for 0.080 to 335 K’

# import pickle

# output_file_folder = ’calibrations/’
output_file_folder = ’’
output_file_name = ’PH_fit_PDFv2_SCM1_2017’
pickled_file = output_file_folder + output_file_name + ’.p’
```

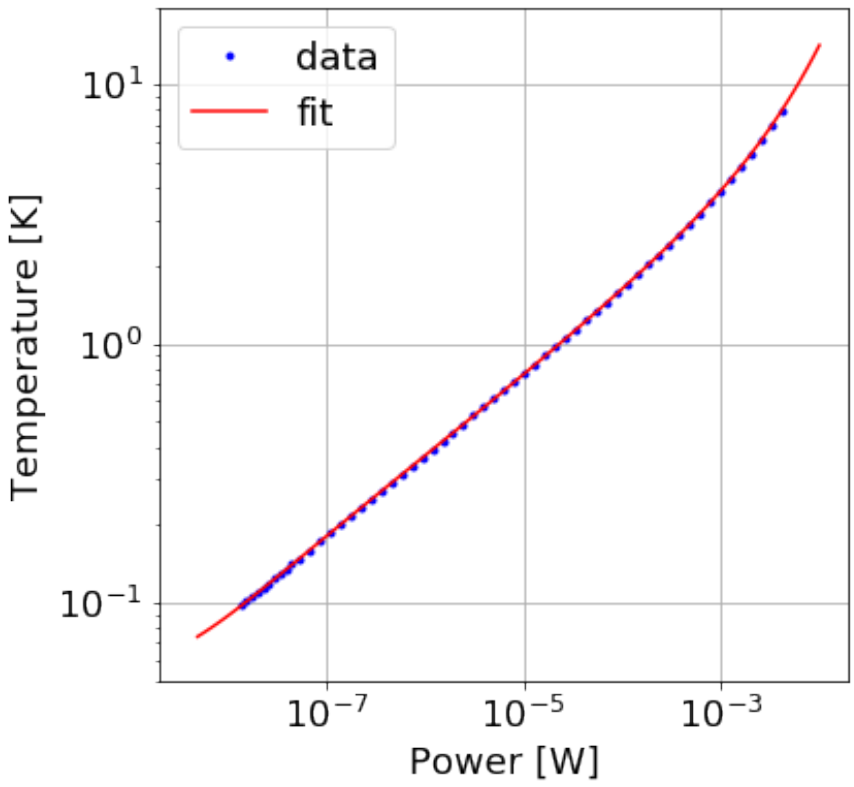

Figure 5: Variation of calorimeter platform temperature versus platform heater power. Fit to $T(P)$ shown as solid line. Original data shown as points.

```
with open(pickled_file, "wb") as f:
    pickle.dump(PH_fit, f)
```

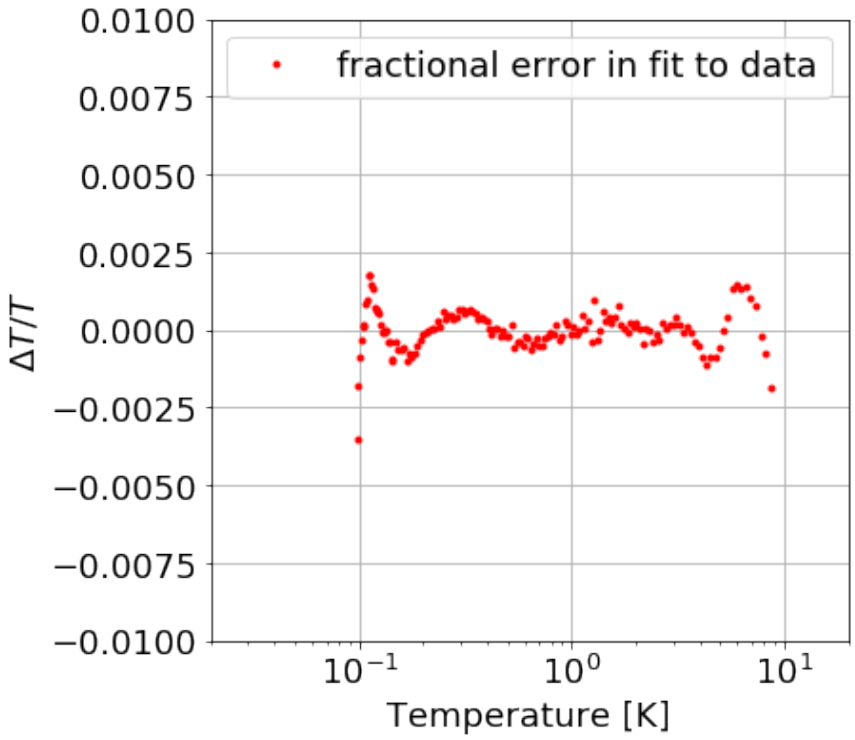

Figure 6: Comparison of $T\,(R_{CT})$ and $T\,(P_{PH})$ calibrations, expressed as the fractional deviation of $T\,(P)$ from original $T\,(R_{CT})$ calibration.

4 R(B,T) calibration

4.1 magnetic field dependence of R(T)

Before fitting the field dependence, we need to calculate Chebyshev coefficients for zero field and non-zero field in a self-consistent way. To do this, we use the (assumed) field-independent calculation of T from platform heater power P_{PH} for all field values, including zero field.

The first step is to set the temperature domain over which the fit will apply. In this example, our data for platform heater power sweeps covers the range 97.5 mK to 8.89 K and we choose 50 mK to 10 K as the T domain for our Chebyshev fits of $R\,(P_{PH})$ to $T\,(P_{PH})$.

The second step is to apply the fit for each magnetic field for which we have $R\,(P)$ data in the same manner as done previously for our zero field data. The results are shown below in Fig. 7 for a Cernox® 1010 for a series of representative magnetic fields between 0 and 18 tesla (180 kG).

4.2 magnetic field dependence of c_i

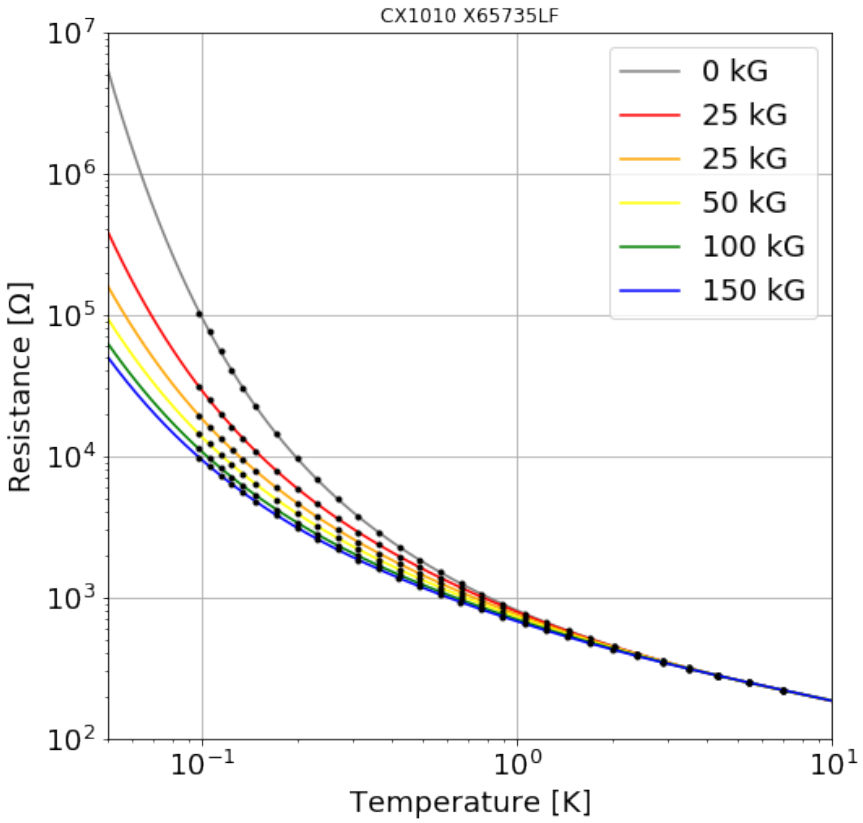

Figure 7: R(T) dependence of a Cernox® 1010 resistive thermometer in a series of applied magnetic fields from 0 to 150 kGauss (15 tesla). A subset of the original data points — converted from (R,P) to (R,T) — are shown as dots. The corresponding R(T) Chebyshev fits are shown as solid lines.

Next, we generate generated arrays of $c_i\,(B)$ for each Chebyshev coefficient. In this case, we restrict ourselves to 6 coefficients — c_0 through c_5 — since higher order fits either did not converge for all field values and/or introduced artificial discontinuities in the higher order coefficients as a function of magnetic field. The results are shown below in Fig. 8.

It is instructive to re-plot these results as the fractional deviation $y_i\,(B) = \frac{(c_i(B) - c_i(0))}{c_i(0)}$ of each coefficient $c_i\,(B)$ from its zero field value $c_i\,(0)$, as shown in Fig. 9 and described in Eq. 3. We see that the field dependences of $y_i\,(B)$ have approximately the same functional form for all six coefficients and appear to converge to a common curve with increasing index value i.

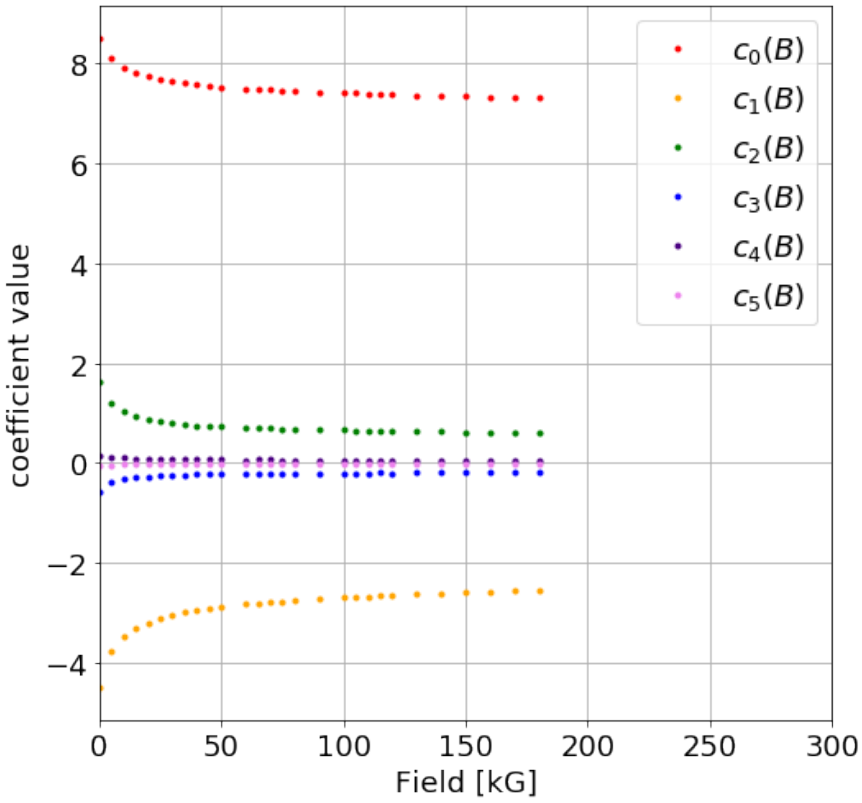

Figure 8: Magnetic field dependence of the Chebyshev coefficients $c_i\,(B)$ used to fit the temperature dependence of a resistive thermometer (CX1010) shown in Fig. 7 , as described in Eq. 1. The zero-field values of the coefficients alternate in sign and converge in absolute value. Each coefficient initially exhibits a strong magnetic field dependence; each appears to be approaching an asymptotic limit at high field.

5

The data and code used to process the files and generate this and the preceeding graphs in this section is attached to the figure. Click on the </> Code button to view and then run the Jupyter notebook containing the Python code.

5.1 Padé fit to $c_i\,(B)$

To narrow down our choice of P and Q in Eq.3 and , in addition, to determine initial values for $\gamma_{i,p}$ and $\kappa_{i,q}$coefficients in Eq.3 , let's generate Pade approximants (also known as rational polynomials) for functions resembling the variation of $y_i\,(B)$ with respect to magnetic field B, then use those functional forms as our first attempts at modeling $y_i\,(B)$.

Padé approximants to a function can be generated from a power series expansion of that function; the 2D array of rational Padé approximants generated for various choices of P and Q is known as a Padé table. For an example, see the sample Padé table for the exponential function e^z posted on Wikipedia.

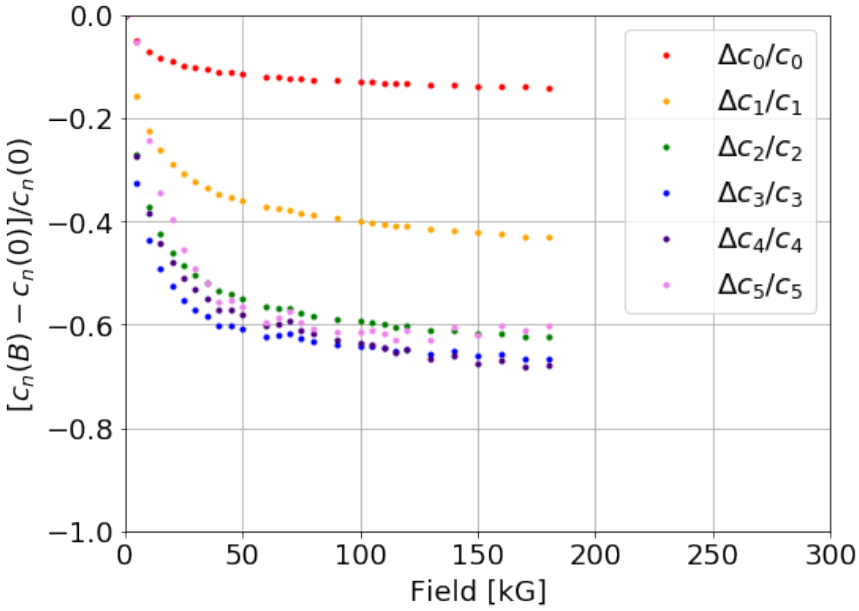

Figure 9: fractional deviations of Chebyshev coefficients $c_i\,(B)$ from zero field values $c_i\,(0)$ for a Cernox® CX1010 resistive thermometer. This sensor has a large magnetoresistance at low temperature but the magnetic field dependence of the Chebyshev fit coefficients has approximately the same functional form for each coefficient.

Although Pade approximants can be calculated analytically, we will instead use the mpmath Python module to generate both the power series expansion and the corresponding Padé series.

5.1.1 mpmath package

Here is an example of how to use the mpmath package to generate a series expansion for e^z for $z < 1$:

```
# import mpmath package and set some common parameters
import mpmath as mp
mp.dps = 15; mp.pretty = True
one = mp.mpf(1)

# define function
def f(x):
    return mp.exp(x)

# generate Taylor series

f_Taylor = mp.taylor(f, 0, 4)
mp.nprint(f_Taylor)
```

with resulting coefficient values

```
[1.0, 1.0, 0.5, 0.166667, 0.0416667]
```

where 0.16667 = 1/6 and 0.0416667 = 1/24.

The corresponding Taylor series expansion is thus

$$\exp(z) \approx 1 - z + \left(\frac{1}{2!}\right) z^2 - \left(\frac{1}{3!}\right) z^3 + \left(\frac{1}{4!}\right) z^4 + ... \tag{10}$$

as expected.

We now use this Taylor series expansion to generate a P = 2, Q= 2 Padé approximant:

```
p_series,q_series = mp.pade(f_Taylor, 2, 2)
mp.nprint(p_series)
mp.nprint(q_series)
```

with output

```
[1.0, 0.5, 0.0833333]
[1.0, -0.5, 0.0833333]
```

where 0.0833333 = 1/12.

The corresponding Padé series expansion is

$$e^z = \frac{\left(1 + \frac{1}{2}z + \frac{1}{12}z^2 + ...\right)}{\left(1 - \frac{1}{2}z + \frac{1}{12}z^2 + ...\right)} \tag{11}$$

as expected.

5.1.2 functional approximation

For the CX1010 thermometer shown above, the $y_i\,(B)$ resemble exponential functions beginning at 0 for B = 0 and approaching an asymptotic value a_i, where $a_i < 0$:

$$y_i(B) = a_i \left(e^{-B/B_0} - 1\right) + ... \tag{12}$$

plus a possible linear term not included in the above expression. Here B_0 is scaling factor chosen so that $\frac{B}{B_0}$ is dimensionless.

As before, we first expand this function in terms of a Taylor series (temporarily setting the multiplicative term $a_i = 1$)

```
def g(x):
    return mp.exp(-x) - 1

g_Taylor = mp.taylor(g, 0, 6)
mp.nprint(g_Taylor)
```

with result

```
[0.0, -1.0, 0.5, -0.166667, 0.0416667, -0.00833333, 0.00138889]
```

where, as before, we assign $0.0833333 = 1/12$ and $0.0013889 = 1/720$.

Re-expressing this Taylor series expansion as a P=2, Q=2 Padé approximant

```
p_series_2,q_series_2 = mp.pade(g_Taylor, 2, 2)
mp.nprint(p_series_2)
mp.nprint(q_series_2)
```

yields

```
[0.0, -1.0, 1.11022e-16]
[1.0, 0.5, 0.0833333]
```

and, after rounding off to nearest fractions,

$$(e^{-z} - 1)\Big|_{2,2} \approx \frac{\left(0 - z + 0z^2\right)}{\left(1 + \frac{1}{2}z + \frac{1}{12}z^2\right)} \tag{13}$$

For $z < 1$. Alternatively, if we need additional terms in our expansion, we might choose to re-express the same Taylor series expansion as a higher order P=3, Q=3 Padé approximant

```
p_series_3, q_series_3 = mp.pade(g_Taylor, 3, 3)
```

which, after rounding off to nearest fractions, yields

$$(e^{-z} - 1)\Big|_{3,3} \approx \frac{\left(0 - z + 0z^2 - \frac{1}{60}z^3\right)}{\left(1 + \frac{1}{2}z + \frac{1}{10}z^2 + \frac{1}{120}z^3\right)} \tag{14}$$

Comparing Eqs.13 and 14 to Eq. 3 , we notice that in both expansions, $\kappa_{i,0}$ and $\kappa_{i,2}$ equal zero and, in addition, $\gamma_{i,q} > 0$ for all q. This suggests that the Padé approximants for $y_i(B)$ should likewise set the coefficients for the numerator in Eq. 3 equal to zero for all even powers of B and require the coefficients for the denominator to be positive values. It further suggests that in addition to requiring that $\kappa_{i,0} = 0$, $\kappa_{i,2} = 0$, and $\gamma_{i,0} = 1$, we should set the initial values of the remaining coefficients in the numerator as follows: $\kappa_{i,1} = -|a_i|$, $\kappa_{i,3} = -\frac{1}{60}|a_i|$ — where a_i represents the approximate asymptotic value of y_i at high field — and the initial values of the coefficients in the denominator as $\gamma_{i,1} = \frac{1}{2}$, $\gamma_{i,2} = \frac{1}{10}$, etc.

Before proceeding, however, we still need to account for the approximately linear term in B in the high field limit. One way to do this while still using the results of Eq. 14 is to drop the highest order term in the numerator. Our proposed function then becomes

$$y_i(B) = \left.\frac{\sum_{p=0}^{P} \kappa_{i,p} B^p}{\sum_{q=0}^{Q} \gamma_{i,q} B^q}\right|_{CX1010} = \frac{\kappa_{i,1}(B/B_0) + \kappa_{i,3}(B/B_0)^3}{1 + \gamma_{i,1}(B/B_0) + \gamma_{i,2}(B/B_0)^2} \tag{15}$$

where B_0 is a scaling factor chosen to make $\frac{B}{B_0}$ dimensionless.

5.1.3 Python code: Padé fit to $c_0(B)$

define Python function for Eq. 15:

```
def scaled_Pade_dependence_32(field, kappa1, kappa3, gamma1, gamma2):
    numerator = ( kappa1*(field) + kappa3*(field)**3)
    denominator = (1 + gamma1 * (field) + gamma2 * (field)**2)
    return numerator/denominator
```

Non-linear curve fit of $y_0(B)$ data to Python function:

```
# specify initial values for fit

a_0 = 0.1  # this is the asymptotic value (absolute value)
initial_guess = np.array([-1* a_0,  -1/60.0  * a_0, 1/2, 1/10])
boundary_values = ([ -np.inf, -np.inf,0, 0], [ 0, 0, np.inf, np.inf ])

# scale the field values

# B_scale = 0.1 # if data is originally in tesla. Converts to kG.
B_scale = 1 # if data is originally in kilogauss
scaled_field = B_field / B_scale

# execute curve fit for y_0(B) data array (scaled_coef_0)

ScaledPadeFit_0, ScaledPadeCovariance_0  = curve_fit(
    scaled_Pade_dependence_32, scaled_field, scaled_coef_0,
    p0 = initial_guess, bounds = boundary_values)
```

Evaluate fit at series of magnetic field values:

```
# generate a series of field values
B_test = np.linspace(0, 360, 2000) / B_scale
```

```
# calculate y(B) using the function above (scaled_Pade_dependence_32)
scaled_Pade_0 = scaled_Pade_dependence_32(B_test, ScaledPadeFit_0[0],ScaledPadeFit_0[1] , ScaledPadeFit
```

Generate plot of $c_0\,(B)$ data and fit:

```
# plot the results

plt.figure(figsize = (8,8))
plt.plot(B_test * B_scale , scaled_Pade_0, 'r', label = 'Padé fit')
plt.plot(B_field, scaled_coef_0, 'b.', label = 'data')
plt.xlabel('Field [kG]', fontsize = 18)
plt.ylabel('$y_0(B)$', fontsize = 18)
plt.xlim(-20, 300)
# plt.ylim(100, 1E7)
plt.legend(loc = 'best', fontsize = 18)
plt.grid(True)
```

The results are shown below for $y_0(B) = (c_0(B) - c_0(0))/c_0(0)$, the fractional deviation of Chebyshev coefficient $c_0\,(B)$ from its zero field value (as specified in Eq. 1).

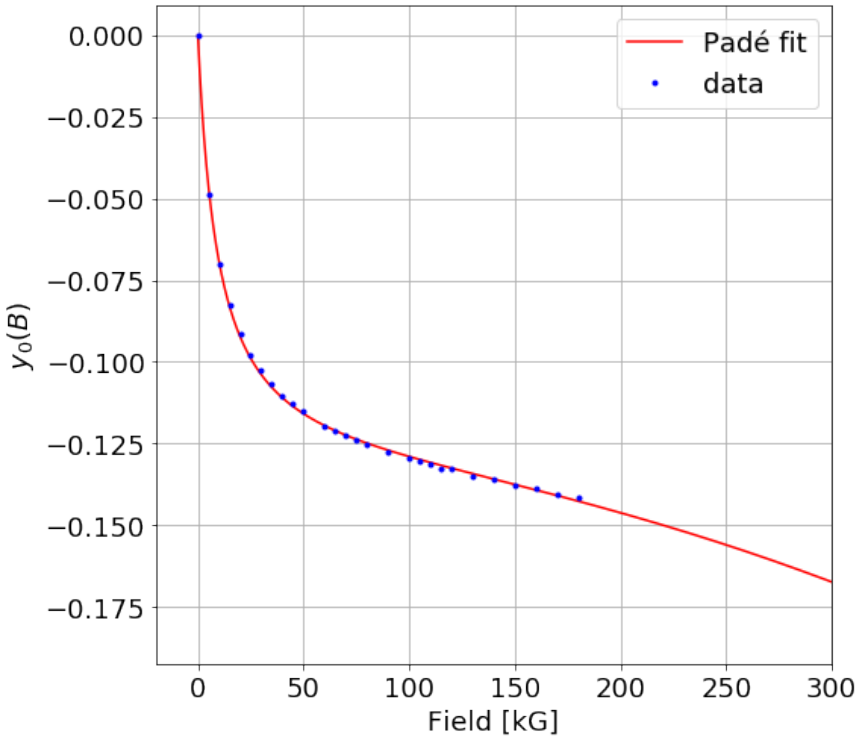

Figure 10: Padé fit to magnetic field dependence of $y_0(B)$ for a CX1010 resistive thermometer. Here $y_0(B) = (c_0(B) - c_0(0))/c_0(0)$, where $c_0(B)$ is the (magnetic-field-dependent) value of the first coefficient in the Chebyshev polynomial fit of $R(T)$ in Eq. 1.

Evaluating the coefficients for significance,

```
ScaledPadeError_0 = np.sqrt(np.diag(ScaledPadeCovariance_0))
print('kappa_1 =' , '{:.2g}'.format(ScaledPadeFit_0[0]), '±', '{:.2g}'.format(ScaledPadeError_0[0]))
print('kappa_3 =' , '{:.2g}'.format(ScaledPadeFit_0[1]), '±', '{:.2g}'.format(ScaledPadeError_0[1]))
print('gamma_1 =' , '{:.2g}'.format(ScaledPadeFit_0[2]), '±', '{:.2g}'.format(ScaledPadeError_0[2]))
print('gamma_2 =' , '{:.2g}'.format(ScaledPadeFit_0[3]), '±', '{:.2g}'.format(ScaledPadeError_0[3]))
```

with results

```
kappa_1 = -0.0139 ± 0.00036
kappa_3 = -3.93e-08 ± 1.7e-08
gamma_1 = 0.101 ± 0.0043
gamma_2 = 5.87e-16 ± 3e-05
```

Since the $\gamma_{0,2}B^2$ term in the denominator of Eq. 15 does not significantly differ from zero, we should try dropping that term.

Running the fit again for the modified Padé expansion specified by Eq.16 below (omitting the $\gamma_{0,2}B^2$ term from Eq. 15)

$$y_i(B) = \left.\frac{\sum_{p=0}^{P} \kappa_{i,p} B^p}{\sum_{q=0}^{Q} \gamma_{i,q} B^q}\right|_{CX1010} = \frac{\kappa_{i,1}(B/B_0) + \kappa_{i,3}(B/B_0)^3}{1 + \gamma_{i,1}(B/B_0)} \tag{16}$$

we modify the Python code as follows:

```
def scaled_Pade_dependence_31(field, kappa1, kappa3, gamma1):
    numerator = ( kappa1*(field) + kappa3*(field)**3)
    denominator = (1 + gamma1 * (field))
    return numerator/denominator

a_0 = 0.1  # this is the asymptotic value (absolute value)
initial_guess = np.array([-1* a_0,  -1/60.0  * a_0, 1/2])
boundary_values = ([ -np.inf, -np.inf,0], [ 0, 0, np.inf ])

B_scale = 1
scaled_field = B_field / B_scale

ScaledPadeFit_0, ScaledPadeCovariance_0  = curve_fit(
    scaled_Pade_dependence_31, scaled_field, scaled_coef_0,
    p0 = initial_guess, bounds = boundary_values)
```

Evaluating the remaining fit coefficients in the same way as before , we find the same final numerical values, thus confirming that the γ_{i2} term is unnecessary over this field range (at least for c_0). The plot of this new fit is the same as that already shown in Fig. 10.

```
kappa_1 = -0.0139 ± 0.00022
kappa_3 = -3.93e-08 ± 3.4e-09
gamma_1 = 0.101 ± 0.002
```

The Padé fit appears valid over the full range of the data (0 to 18 tesla). The change in curvature above 18 T in the extrapolated fit seems unlikely but data at higher fields would be needed to be sure. Higher field

data might also allow the use of additional terms in the denominator and better constrain the high field extrapolation.

5.1.4

5.1.5 Complete Padé fit

We fit the remaining coefficients to Eq. 16 in the same way as before, with one difference: we use the final numerical values for the Padé coefficients for the previous fit as the initial values for the Padé coefficients for the new fit. Thus the coefficient values given above for y_0 become the initial values for the fit to y_1, the final values for y_1 become the initial values for y_2, etc. This approach is justified here because each $y_i(B)$ curve is similar to the previous curve for $y_{i-1}(B)$, as shown in Fig. 9.

In the following figures, we show the results for $y_1(B)$ through $y_5(B)$. With the exception of $\kappa_{5,3}$, all of the coefficients are significantly different from zero (which we therefore set equal to zero).

In general, the fits agree with the data over the entire field range but in the case of $y_5(B)$, the fit begins to diverge from the data above 125 kG (12.5 tesla). Since this is the smallest of the coefficients in the orthogonal Chebyshev expansion, the deviation of the fit from the data will be relatively unimportant in the calculation of $R(T, B)$. Additional data at higher fields should allow the inclusion of higher order terms in this fit (and those for the other y_i coefficients), improving the fit to the high field behavior.

Here is the fit to $y_1(B)$:

with coefficients

```
kappa_1 = -0.0469 ± 0.00073
kappa_3 = -1.12e-07 ± 1e-08
gamma_1 = 0.111 ± 0.0021
```

Here is the fit to $y_2(B)$:

with coefficients

```
kappa_1 = -0.0902 ± 0.0012
kappa_3 = -1.17e-07 ± 1.5e-08
gamma_1 = 0.144 ± 0.0023
```

Here is the fit to $y_3(B)$:

with coefficients

```
kappa_1 = -0.125 ± 0.0018
kappa_3 = -8.34e-08 ± 1.8e-08
gamma_1 = 0.187 ± 0.003
```

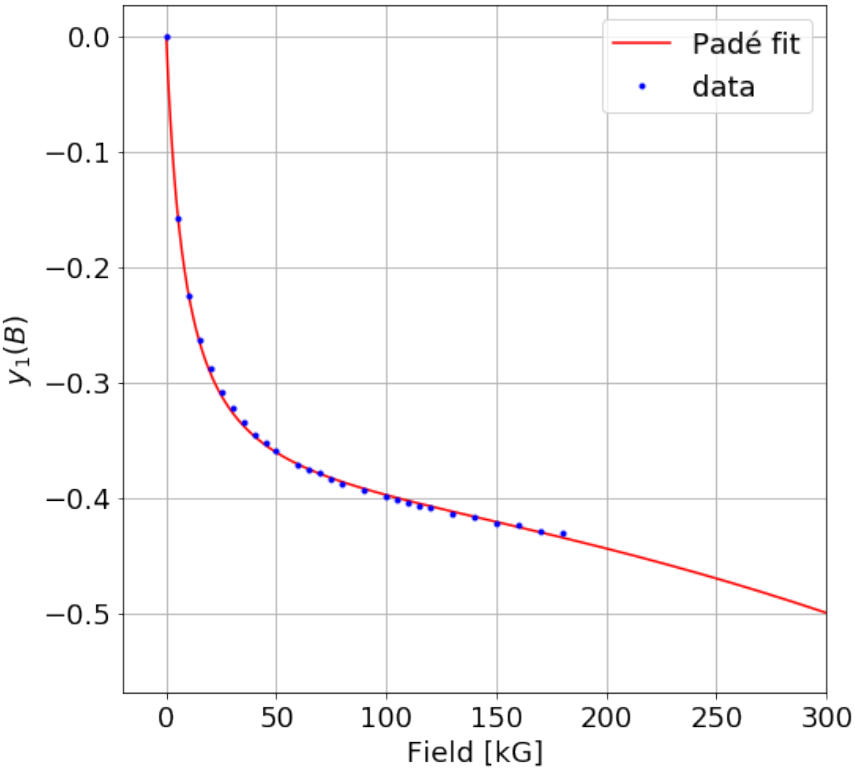

Figure 11: Padé fit to $y_1\,(B)$ for the CX1010 resistive thermometer data shown in Fig. 7. See Eqs. 1, 3, and 16 for details.

Here is the fit to $y_4\,(B)$:

with coefficients

```
kappa_1 = -0.0894 ± 0.0022
kappa_3 = -2.1e-07 ± 2.9e-08
gamma_1 = 0.134 ± 0.004
```

and finally, here is the fit to $y_5\,(B)$:

with coefficients

```
kappa_1 = -0.047 ± 0.0057
kappa_3 = -6.52e-27 ± 8.5e-08
gamma_1 = 0.0671 ± 0.011
```

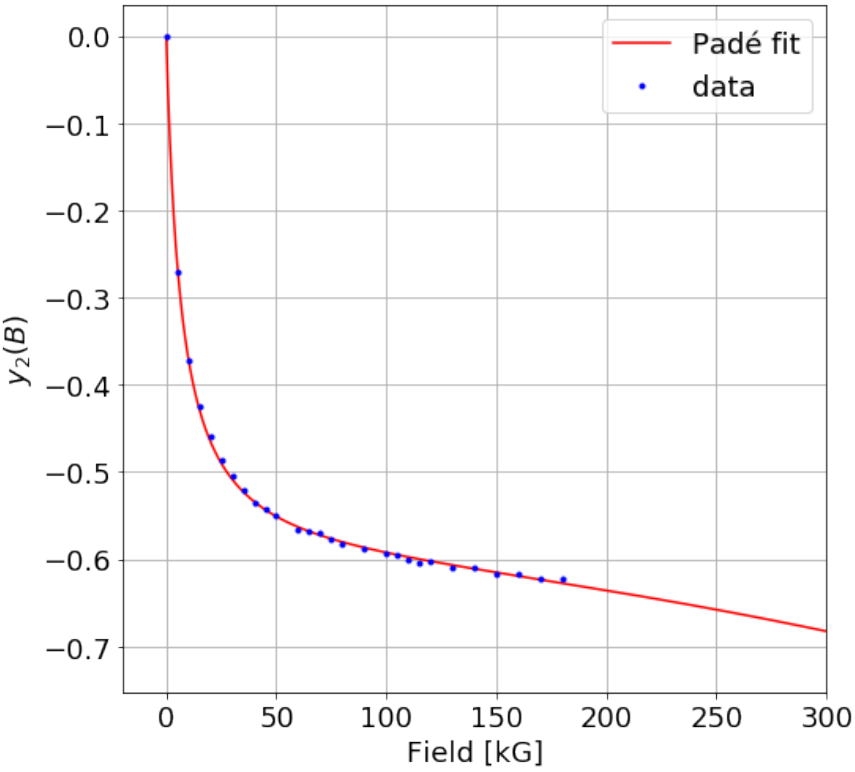

Figure 12: Padé fit to $y_2\,(B)$ for the CX1010 resistive thermometer data shown in Fig. 7. See Eqs. 1, 3, and 16 for details.

As noted above, the $\kappa_{5,3}$ value is not significantly different from zero and is therefore replaced by zero in all calculations of c_5.

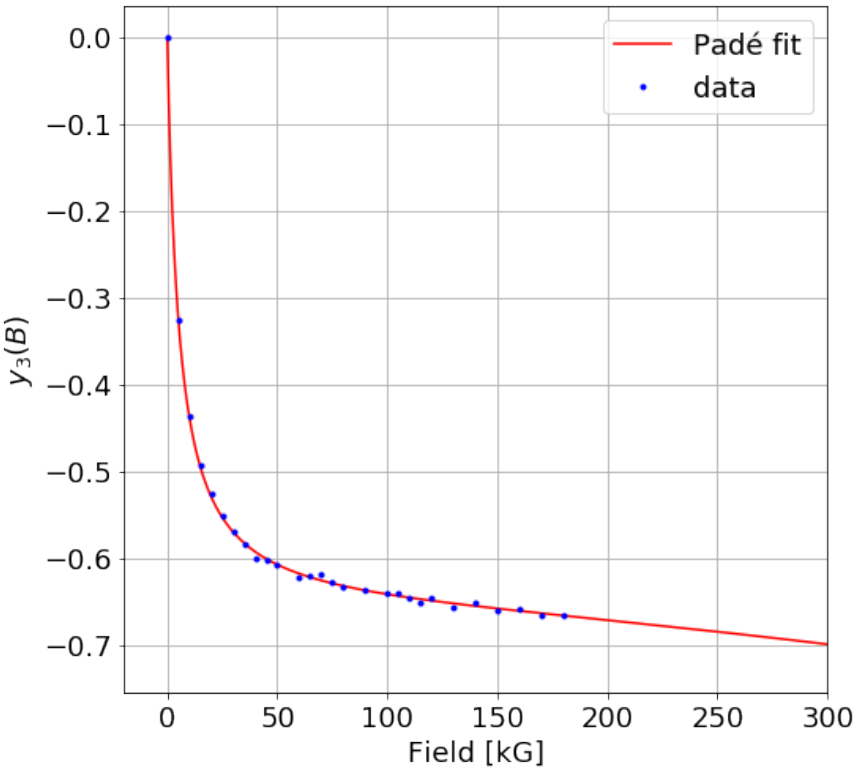

Figure 13: Padé fit to $y_3\,(B)$ for the CX1010 resistive thermometer data shown in Fig. 7. See Eqs. 1, 3, and 16 for details.

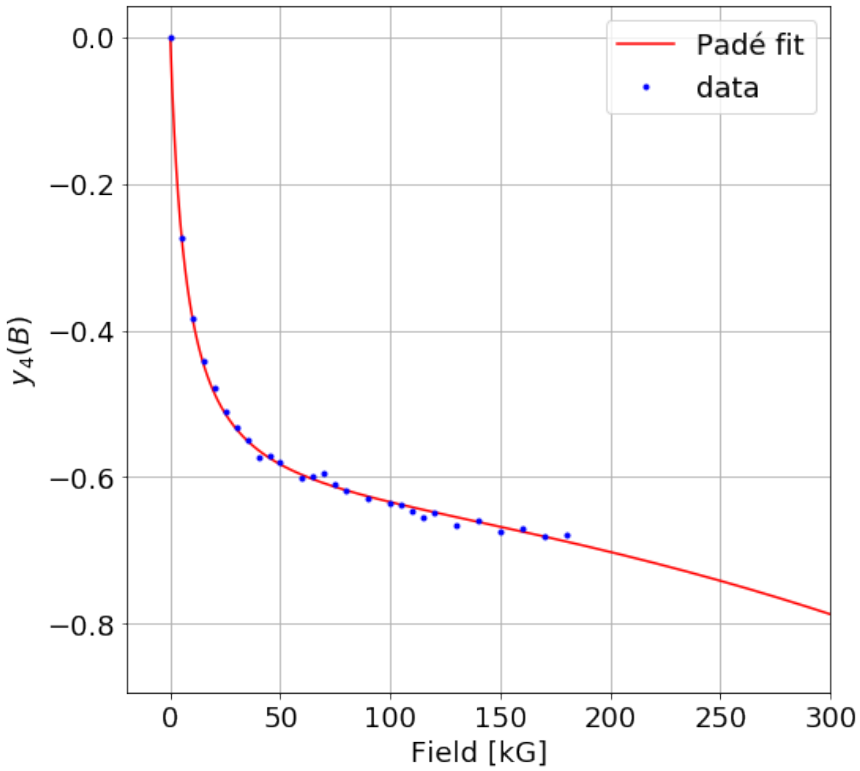

Figure 14: Padé fit to $y_4\,(B)$ for the CX1010 resistive thermometer data shown in Fig. 7. See Eqs. 1, 3, and 16 for details.

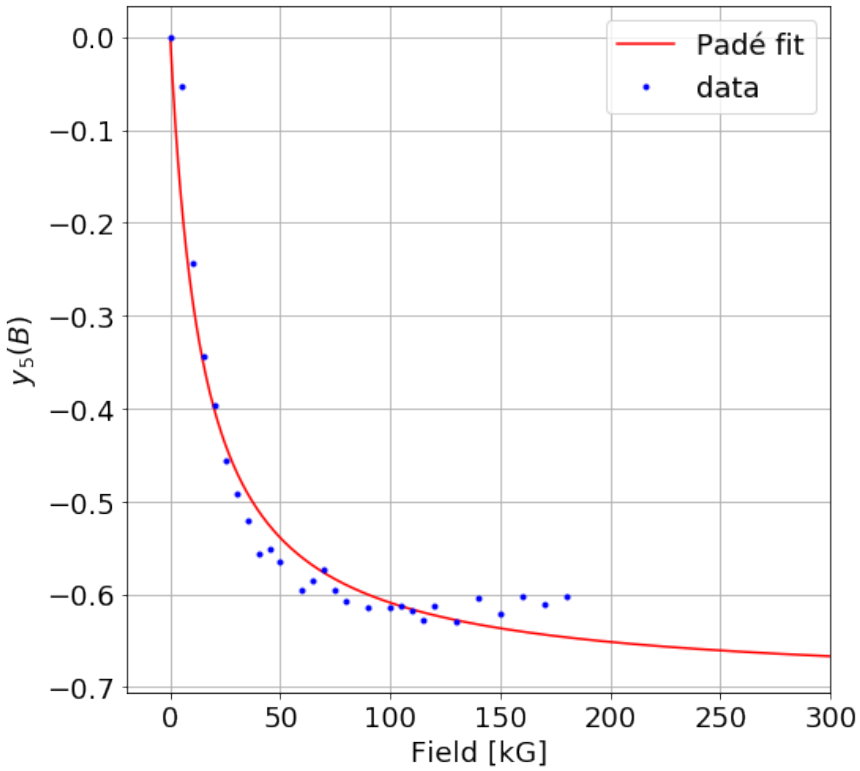

Figure 15: Padé fit to $y_5\,(B)$ for the CX1010 resistive thermometer data shown in Fig. 7. See Eqs. 1, 3, and 16 for details.

6 Calculation of T from R, B

We now have what we need to calculate temperature from measurements of resistance in a magnetic field but we need to put the calibration coefficients into a more useful form that won't require us to first rerun the calibration each time we want to calculate the temperature.

6.1 mathematical representation

For this purpose, it is convenient to store the calibration coefficients $\kappa_{i,p}$ and $\gamma_{i,q}$ in the form of 2D matrices $\boldsymbol{\kappa}$ and $\boldsymbol{\gamma}$ and $c_i\,(0)$ as the 1D matrix (array) $\mathbf{c}_0$.

That is, for $i = 0,\ 1,\ ...N\ ,\ \ p\ =\ 0,\ 1,\ ...P$, and $q\ =\ 0,\ 1,\ ...Q,$ define the following matrices:

$$\kappa_{i,p} = \begin{pmatrix} \kappa_{0,0} & \kappa_{0,1} & \cdots & \kappa_{0,P} \\ \kappa_{1,0} & \kappa_{1,1} & \cdots & \kappa_{1,P} \\ . & . & \cdots & . \\ . & . & \cdots & . \\ . & . & \cdots & . \\ \kappa_{N,0} & \kappa_{N,1} & \cdots & \kappa_{N,P} \end{pmatrix} \tag{17}$$

$$\gamma_{i,q} = \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} \\ \gamma_{1,0} & \gamma_{1,1} \\ . & . \\ . & . \\ . & . \\ \gamma_{N,0} & \gamma_{N,Q} \end{pmatrix} \tag{18}$$

and

$$\mathbf{c}_0^{\mathrm{T}} = \begin{pmatrix} c_0(0) & c_1(0) & \ldots & c_N(0) \end{pmatrix} \tag{19}$$

with $\kappa_{i,0}\ = 0$ and $\gamma_{i,0} = 1$ for all i .

If, in addition, we create the P x 1 matrices $\mathbf{b}_\kappa$ and $\mathbf{n}$ and Q x 1 matrices $\mathbf{b}_\gamma$ and $\mathbf{d}$, where

$$\mathbf{b}_\kappa = \begin{pmatrix} B^0 \\ B^1 \\ B^2 \\ \cdot \\ \cdot \\ \cdot \\ B^P \end{pmatrix} \tag{20}$$

$$\mathbf{b}_\gamma = \begin{pmatrix} B^0 \\ B^1 \\ \cdot \\ \cdot \\ \cdot \\ B^Q \end{pmatrix} \tag{21}$$

$$\mathbf{n} = \kappa \mathbf{b}_\kappa = \begin{pmatrix} \kappa_{0,0} & \kappa_{0,1} & \cdots & \kappa_{0,P} \\ \kappa_{1,0} & \kappa_{1,1} & \cdots & \kappa_{1,P} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \kappa_{N,0} & \kappa_{N,1} & \cdots & \kappa_{N,P} \end{pmatrix} \begin{pmatrix} 1 \\ B \\ \cdot \\ \cdot \\ \cdot \\ B^P \end{pmatrix} = \begin{pmatrix} n_0 \\ n_1 \\ \cdot \\ \cdot \\ \cdot \\ n_\mathrm{N} \end{pmatrix} \tag{22}$$

and

$$\mathbf{d} = \gamma \mathbf{b}_\gamma = \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,Q} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,Q} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \gamma_{N,0} & \gamma_{N,1} & \cdots & \gamma_{N,Q} \end{pmatrix} \begin{pmatrix} 1 \\ B \\ \cdot \\ \cdot \\ \cdot \\ B^Q \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \cdot \\ \cdot \\ \cdot \\ d_\mathrm{N} \end{pmatrix} \tag{23}$$

then the magnetic field dependent Chebyshev coefficients $c_i\,(B)$ in Eq. 1 become, in array form,

$$\mathbf{c} = {\mathbf{c}_0}^\mathrm{T} \mathbf{z} \tag{24}$$

where

$$\mathbf{z} = \begin{pmatrix} 1 + y_0(B) \\ 1 + y_1(B) \\ . \\ . \\ . \\ 1 + y_N(B) \end{pmatrix} = \begin{pmatrix} 1 + \frac{n_0}{d_0} \\ 1 + \frac{n_1}{d_1} \\ . \\ . \\ . \\ 1 + \frac{n_N}{d_N} \end{pmatrix} \tag{25}$$

6.2 Python implementation

Here is an example of how to construct the coefficient matrices for N = 5, P = 3, and Q = 1

```
# example of brute force assignment of values to matrices

N_limit = 5
P_limit = 3
Q_limit = 1

# construct kappa matrix
kappa_shape = (N_limit + 1, P_limit + 1)

# set all values of matrix to zero to start
kappa_matrix = np.zeros(kappa_shape)

# construct gamma matrix
gamma_shape = (N_limit + 1, Q_limit + 1)

# set all values of matrix to zero to start
gamma_matrix = np.zeros(gamma_shape)

# assign remaining values if non-zero
kappa_matrix[0,1] = -1.39268738e-02
kappa_matrix[0,3] = -3.93219802e-08
# kappa_matrix[1,1] = ...
# kappa_matrix[1,3] = ...
# ...
# kappa_matrix[N_limit, P_limit] = ...

# set all rows in first column of gamma (q = 0) equal to 1
gamma_matrix[:,0] = 1

# assign remaining values if non-zero
# gamma_matrix[0,1] = -1.39268738e-02
# gamma_matrix[1,1] = ..
# ...
# gamma_matrix[N_limit, Q_limit] = ...
```

Once the coefficient matrices have been constructed, we can construct the corresponding field arrays $\mathbf{b}_\kappa$ and $\mathbf{b}_\gamma$ using the function `field_array(field_value, coefficient_matrix)`.

```
def field_array(field_value,coefficient_matrix):
    '''
    create an array ([$B^0$,$B^1$,$B^2$,..., $B^N$]) from field value B
    where N is the number of columns in the coefficient matrix
    '''

    field_power = np.arange(0,np.shape(coefficient_matrix)[1], 1)
    return field_value ** field_power

# example
B = 100 # kG
b_kappa = field_array(B, kappa_matrix)
b_gamma = field_array(B, gamma_matrix)
```

Next, we use Eqs. 24 and 25 to construct a function `field_coefficients` which will create an array of $c_i\,(B)$ values corresponding to a particular value of magnetic field B:

```
def field_coefficients(field_value, c_0_coefficients,kappa_matrix, gamma_matrix ):
    '''generate array of $c_n(B)$ coefficients for R(T) and T(R) '''

    # make a copy of the coefficients so as not to change the original
    c_array = c_0_coefficients.copy()

    # create the field arrays (1, B, B**2, ....)
     B_kappa = field_array(field_value, kappa_matrix)
     B_gamma = field_array(field_value, gamma_matrix)

    numerator_array = np.matmul(kappa_matrix, B_kappa)
    # matrix multiplication of P X 1 matrix (1, B, ...,B**P)
    # by N x P matrix (kappa)
    # yields a N x 1 matrix (1D array)

    denominator_array = np.matmul(gamma_matrix, B_gamma)
    # matrix multiplication of Q X 1 matrix (1, B, ...,B**Q)
    # by N x Q matrix (gamma)
    # yields a N x 1 matrix (1D array)

    y_array = numerator_array / denominator_array
    # reminder: array division is element by element

    z_array = 1 + y_array
    # reminder: here 1 is treated as an array of ones of length y_array

    c_array = c_0_coefficients * z_array
    # reminder: 1D array multiplication is element by element.
    # This is not the dot product np.dot

    return c_array
```

Finally, we define functions that will use these newly calculated coefficients to find

1. the resistance R from the temperature T and magnetic field B
2. the temperature T from the resistance R and the magnetic field B

The function `RandBtoT`calculates the temperature corresponding to a particular resistance value R when measured in a magnetic field of strength B.

```
def RandBtoT(R_values, CT_zero_field_fit, B_value = 0, kappa = kappa_matrix, gamma = gamma_matrix):
    '''calculate T from R and B using Chebyshev_fit'''

    # extract the zero field coefficients from the Chebyshev fit parameters
    c_0_array = CT_zero_field_fit.coef

    # make a copy of the fit parameters to modify
    CT_field_fit = CT_zero_field_fit.copy()

    # replace the zero field coefficients c(0) with the in-field coefficients c(B)
    CT_field_fit.coef = field_coefficients(B_value, c_0_array, kappa, gamma)

    return RtoT(R_values, CT_field_fit)
```

The function `TandBtoR` calculates the expected R value corresponding to a particular T and B. This function is useful when updating the setpoint of a temperature controller that is in ohms. Regular updates as a function of magnetic field provide a way to maintain a constant real temperature during field sweeps.

```
def TandBtoR(T_values,  CT_zero_field_fit, B_value = 0, kappa = kappa_matrix, gamma = gamma_matrix):
    '''calculate R from T and B using Chebyshev_fit'''

    # extract the zero field coefficients from the Chebyshev fit parameters
    c_0_array = CT_zero_field_fit.coef

    # make a copy of the fit parameters to modify
    CT_field_fit = CT_zero_field_fit.copy()

    # replace the zero field coefficients c(0) with the in-field coefficients c(B)
    CT_field_fit.coef = field_coefficients(B_value, c_0_array, kappa, gamma)

    # calculate T(R) using c(B) instead of c(0)
    return TtoR(T_values, CT_field_fit)
```

To instead adjust the setpoint of a temperature controller that is in kelvin, first use `TandBtoR`to find R as before, then use `RandBtoT` (with B set equal to zero) to find the temperature at zero-field that would correspond to that same R value.

Finally, for post-processing of large data sets consisting of arrays of R and B values as a function of time, the function `calculate_T_from_data` will return a corresponding array of T values.

```
def calculate_T_from_data(R_values, B_values, zero_field_fit, kappa_matrix, gamma_matrix):
    '''calculate T from (R,B) data arrays using Chebyshev fit to resistive thermometer'''

    # initialize T array
    T_values = np.zeros(R_values.size)
```

```
    # calculate T from R, B
    for index, temperature in enumerate(T_values):
        temperature = RandBtoT(R_values[index],
        zero_field_fit, B_values[index],
        kappa_matrix, gamma_matrix)

        T_values[index] = temperature

    return T_values
```

6.3 Save calibration information to file

Here we show an example of how to save and reload the calibration data for the CX1010 thermometer named 'CT' (for 'Calibrated Thermometer').

6.3.1 Create calibration array:

```
CT_gamma_matrix = gamma_matrix
CT_kappa_matrix = kappa_matrix

CT_field_calibration = (CT_zero_fit, CT_gamma_matrix, CT_kappa_matrix)
```

6.3.2 Save calibration array as binary file:

```
# provide filename and folder name

# file_folder = 'calibrations/'
file_folder = ''

file_name = 'CX1010_SN_X65735LF_RTB_calibration'
pickled_file = file_folder + file_name + '.p'

# write binary file (opens file, writes file, then closes file)

with open(pickled_file, "wb") as f:
    pickle.dump(CT_field_calibration, f)
```

6.3.3 Reload calibration array:

```
# provide filename and folder name

# file_folder = 'calibrations/'
file_folder = ''

file_name = 'CX1010_SN_X65735LF_RTB_calibration'
pickled_file = file_folder + file_name + '.p'
```

```
# read binary file (opens file, reads file, then closes file)

with open(pickled_file, "rb") as f:
    sensor_cal = pickle.load(f)

# create matrices and arrays from uploaded calibration

sensor_fit, sensor_gamma, sensor_kappa = sensor_cal[0], sensor_cal[1], sensor_cal[2]
```

7 Appendix: linear regression method

One alternative approach to the selection of initial values is to transform the problem into a multiple linear regression problem.

For example, if $P = Q = 2$, then

$$y = \kappa_1 B + \kappa_2 B^2 - y\gamma_1 B - y\gamma_2 B^2 \tag{26}$$

and if we have N data points (B_j, y_j) with $j = 1, 2, ...N$, then

$$y = Ax \tag{27}$$

where A is the NxM matrix

$$A = \begin{bmatrix} B_1 & B_1^2 & -y_1 B_1 & -y_1 B_1^2 \\ B_2 & B_2^2 & -y_2 B_2 & -y_2 B_2^2 \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ B_N & B_N^2 & -y_N B_N & -y_N B_N^2 \end{bmatrix} \tag{28}$$

and x is the coefficient array

$$x = \begin{pmatrix} \kappa_1 \\ \kappa_2 \\ \gamma_1 \\ \gamma_2 \end{pmatrix} \tag{29}$$

To solve this with Python, we can use the SciPy Scientific Python package to compute the Moore-Penrose pseudo-inverse $A^+$of the matrix A, where A^+ is defined the matrix that 'solves' (by least squares minimization) the matrix equation Ax=y, with solution $\overline{x} = A^+ y$. Sample code is provided below:

```
# create elements of matrix A for P = Q = 2
A_matrix = np.column_stack((B_field, B_field **2,  -1 * y_B_values * B_field, -1 * y_B_values * B_field

# solve for Pade coefficients
```

```
x_array = np.linalg.pinv((A_matrix.T).dot(A_matrix)).dot(A_matrix.T.dot(y_B_values))
```

This approach sometimes provides useful initial values for a non-linear fit to Eq. 3 but it has several drawbacks:

1. No guidance is provided as to the appropriate choice of P and Q by this method
2. A different matrix must be constructed and solved for each value of P and Q under consideration
3. There is no way to restrict the range of values and/or signs of the calculated coefficients

The third drawback is the most severe, as divergences in Eq. 3 will occur whenever the denominator term $\sum_{q=0}^{Q} \gamma_{i,q} B^q = 0$ (causing $c_i(B) \rightarrow \infty$). If some of the γ_i coefficients provided by the matrix method are negative, the resulting fit may lead to divergences at (real) values of B within the experimental range of the experiment, spoiling the calculation of $R(T, B)$.

When we tried this matrix method for P = Q = 2 for the CX1010 thermometer data shown above, it returned values that resulted in divergences in all but the first two coefficients. We did not investigate the method further (for example, for P = 3 and Q = 1). In comparison, note that divergences cannot occur at real values of B for the Padé fits to $y_i(B)$ constructed using our original method — in which we modeled our choice of P and Q and the initial values on the Padé expansion of Eq. 12 — because the Padé expansions of $e^{-z}$and $e^{-z} - 1$ only have positive terms in the denominator. Given positive initial values for $\gamma_{i,q}$, we were then able to constrain the fit to only consider positive values through the specification of appropriate `boundary_values`. For further details, see the discussion regarding the setting of parameter boundaries in the Scipy manual entry for `curve_fit` .

8

References

R. Barrio. Algorithms for the integration and derivation of Chebyshev series. *Applied Mathematics and Computation*, 150(3):707–717, mar 2004. doi: 10.1016/s0096-3003(03)00301-1. URL https://doi.org/10.1016%2Fs0096-3003%2803%2900301-1.

Nathanael Fortune, Gayle Gossett, Lydia Peabody, Katherine Lehe, S. Uji, and H. Aoki. High magnetic field corrections to resistance thermometers for low temperature calorimetry. *Review of Scientific Instruments*, 71(10):3825, 2000. doi: 10.1063/1.1310341. URL https://doi.org/10.1063%2F1.1310341.

Andreas Karageorghis. A note on the Chebyshev coefficients of the general order derivative of an infinitely differentiable function. *Journal of Computational and Applied Mathematics*, 21(1):129–132, jan 1988. doi: 10.1016/0377-0427(88)90396-2. URL https://doi.org/10.1016%2F0377-0427%2888%2990396-2.