Plan

19 February 2021 16:30

Project plan:

Title:

"Analysis of electron paramagnetic resonance data for extracting distances in biomolecules."

To measure the distances in biomolecules, the free radicals, called spin-labels are attached to a pair of sites in a molecule. The unpaired electrons of the two labels magnetically interact with one another (as two magnetic dipoles) and from the strength of this interaction one can obtain the distance between them [ref1]. However, in certain scenarios (usually at high magnetic fields) data also contains information on orientations of spin labels that can to be disentangled from the distance using a recently published algorithm [ref 2].

In this project we will scrape the distance measurement datasets from published papers and then apply the algorithm for disentangling distance and orientation. We will then compare the expected distances with the result of the algorithm to find out how it works in a variety of situations.

Tasks:

Weeks 1-2:

- To test tools for extracting data points on graphs produced with known data sets to verify accuracy.
- Decide on the tool we will use for the extraction.

Weeks 2-6:

- Find papers with these distance curves (around 10-20) and save these figures along with their results for their angular and spatial separations.
- Begin gaining familiarity with the data processing algorithm, see if modification/optimisation is required.

Weeks 6-8:

- Use the chosen tool to extract the data from the graphs gathered from the 10-20 sources.
- Apply the sifting algorithm to the above-mentioned graphs and extract the various data.
- Compare this data to the literature values.

Weeks 8-end:

- Based on the data gathered read into its significance based on the literature.
- Write report.

References

[1] Gunnar Jeschke. Deer distance measurements on proteins. Annual review on physical chemistry, 63:419-446, 2012.

[2] Alexey Potapov. Application of spherical harmonics for deer data analysis in systems with a conformational distribution. Journal of Magnetic Resonance, 316:106769, 2020.

Preliminary testing of tools (week 1)

19 February 2021 16:30

Aims:

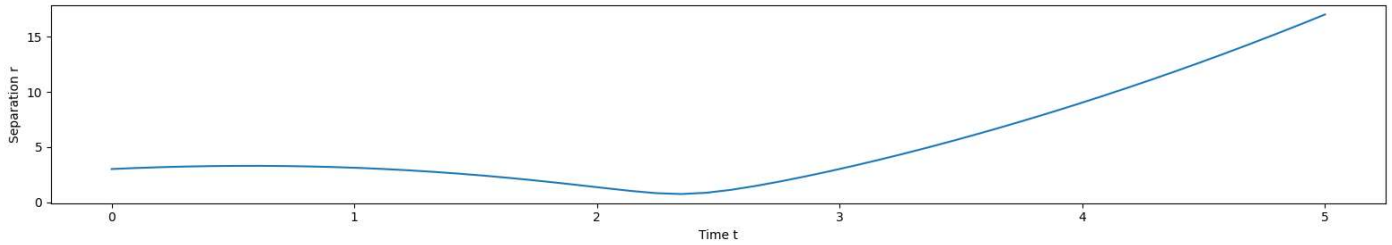
- To find a suitable tool for scraping the data from curves.

Outcome:

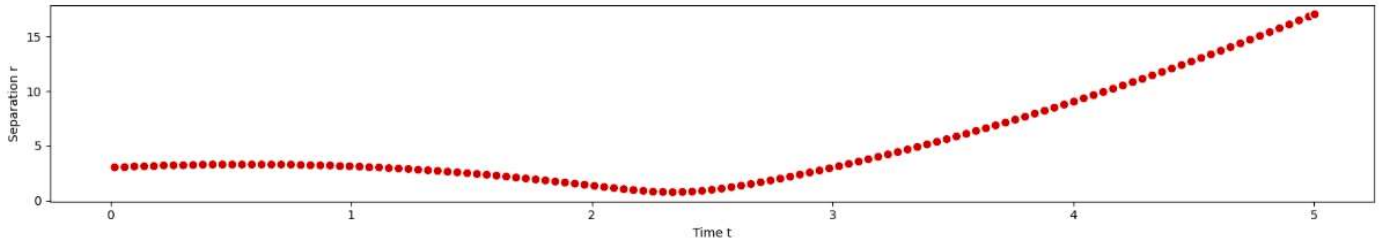
There were two main tool that were tested, this small number is due to many of these software programs for scaping the data required the purchase of licensing to use them.

The first tool we looked into was 'WebPlotDigitizer': https://automeris.io/WebPlotDigitizer/.

This tool is very easy to pick up and start work on immediately. This tool was used to scrape the data from a curve I produced in Python so we could compare the scraped data with the actual data.



This was the image inputted into the 'WebPlotDigitizer' program and once the calibrations were done the points found by the program were plotted as seen below:



A region can be defined within the image and a colour can be isolated. Once the colour/curve is isolated points are placed on this curve and given a coordinate relative to an axis defined by the user.

These coordinates were accurate to 3 significant figures, though with greater tool fluency this can likely be improved.

The second tool looked at was 'DigitizeIt': https://www.digitizeit.xyz/.
This tool was far less intuitive than the previous discussed, though produced similar data to 'WebPlotDigitizer'. However, I could not save to data scraped unless I purchased the license.

A few other tools were looked into though I could find no way to access the. Some offered open access source code, though I was not familiar with the coding language used.

Scraping data from paper with known results (week 2)

19 February 2021 17:12

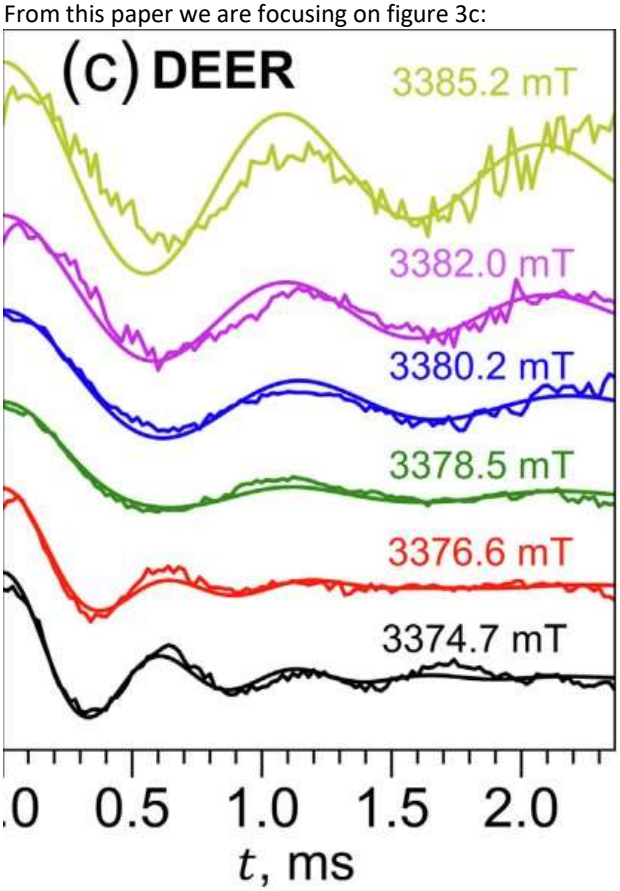
Aims:

- To begin the scraping of data from a figure containing data specific to the subject being studied.
- Begin the gathering of sources for data extraction.

Outcome:

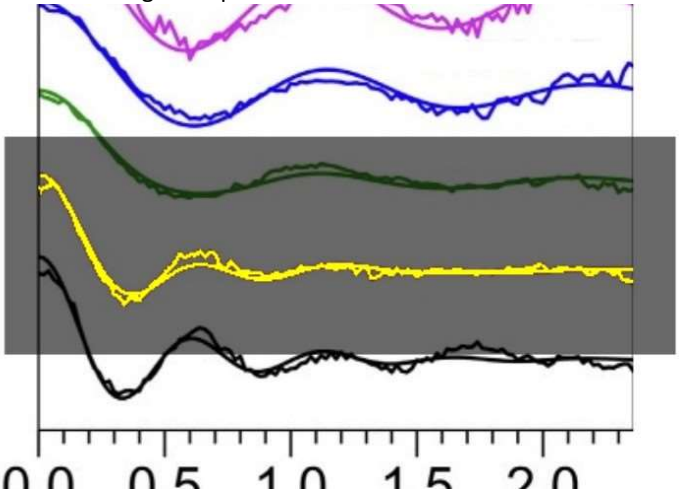
The primary source of focus for these data scraping is from the 2020 paper:
'Application of spherical harmonics for DEER data analysis in systems with a conformational distribution' by Alexey Potapov 2020

From this paper we are focusing on figure 3c:



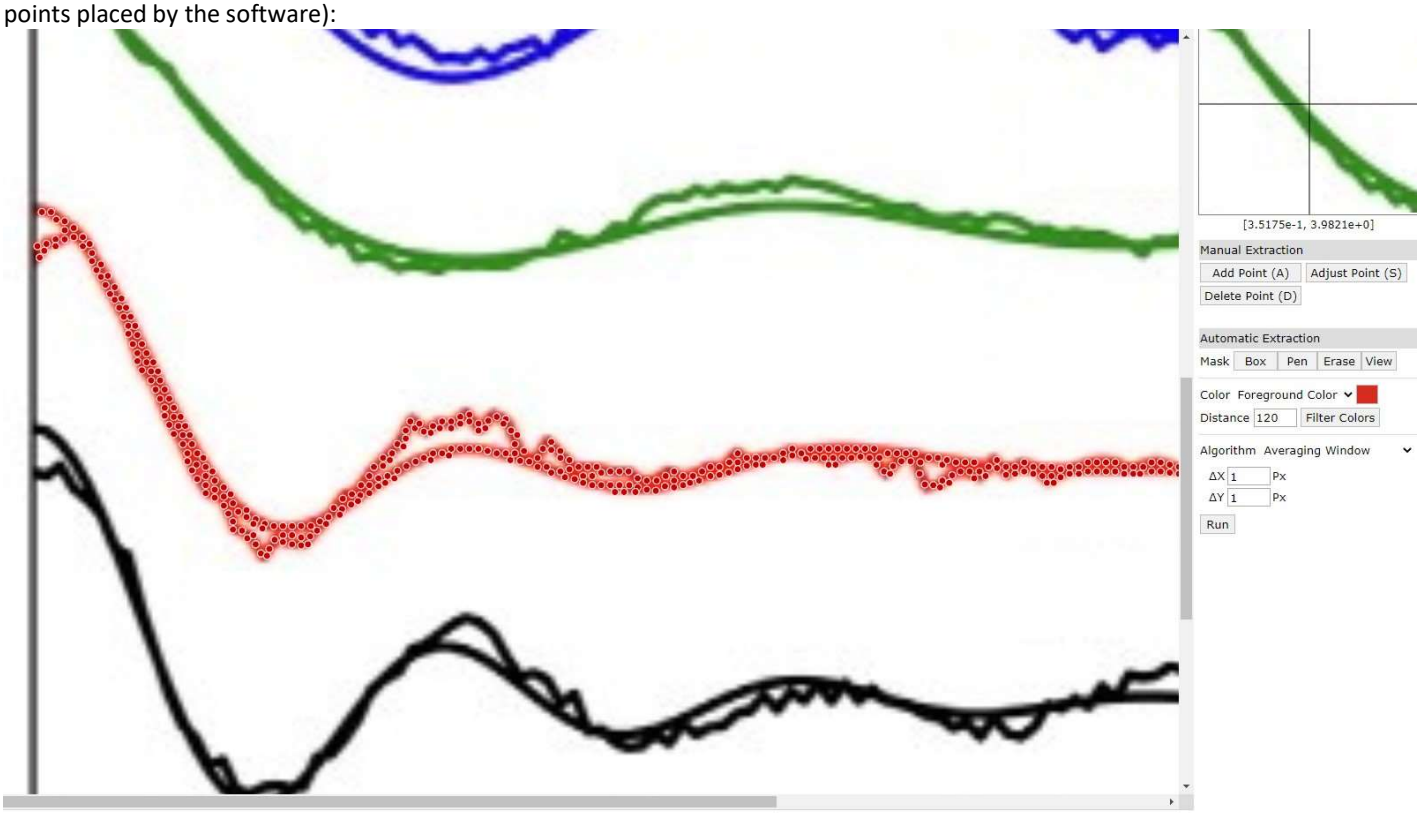
First we edited this image to remove the labels so that data extraction using 'WebPlotDigitizer' would be easier.

The edited image was placed into the software and then each curve was isolated individually:



This was done one at a time re-defining the axes each time. Within the software you can define a precision of the grid used to place the points. For these data extractions we used Δx=Δy = 1 Px.

The data we were gathering was that of the non-smooth curve and so once the points were placed on the curve we had to manually remove those of the smooth model curve (the red spots are the points placed by the software):



The data extracted from these curves was then saved into an Excel file.

Cubic Spline

19 February 2021 17:32

Aims:

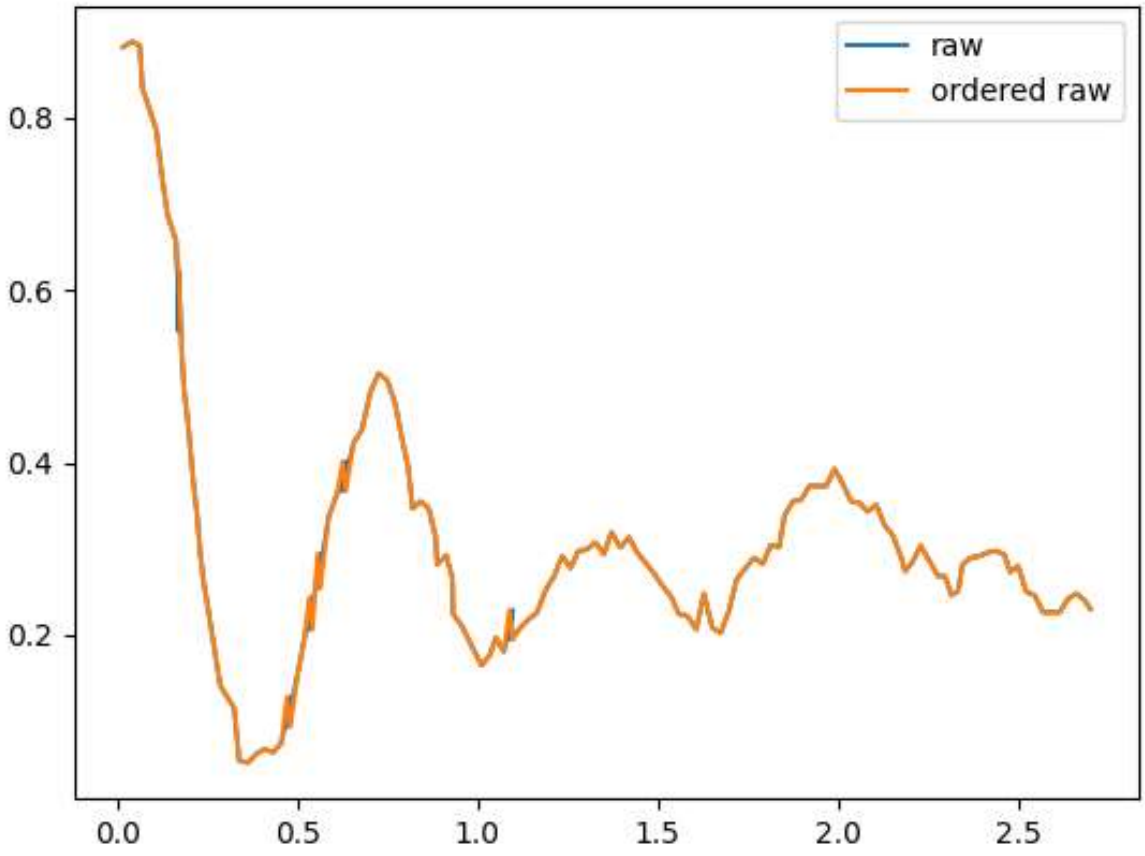
- To perform a cubic spline on the scraped data.

Outcome:

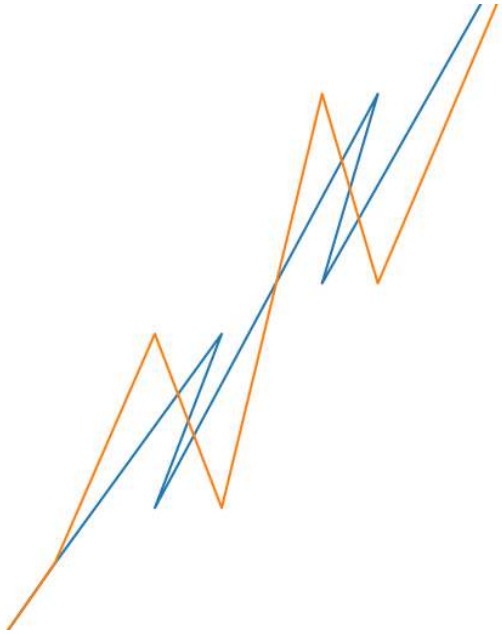
A cubic spline is needed so that the scraped data can be treated as a function and time stamps can be selected.

The Python package that contains the cubic spline function was imported and the function only works for ascending x-values. This was a slight problem as due to the precision we used, $\Delta x=\Delta y= 1$ Px, there were several overlapping points which ran an error when put through the python script, and so the precision was changed to $\Delta x = \Delta y = 2$ Px:

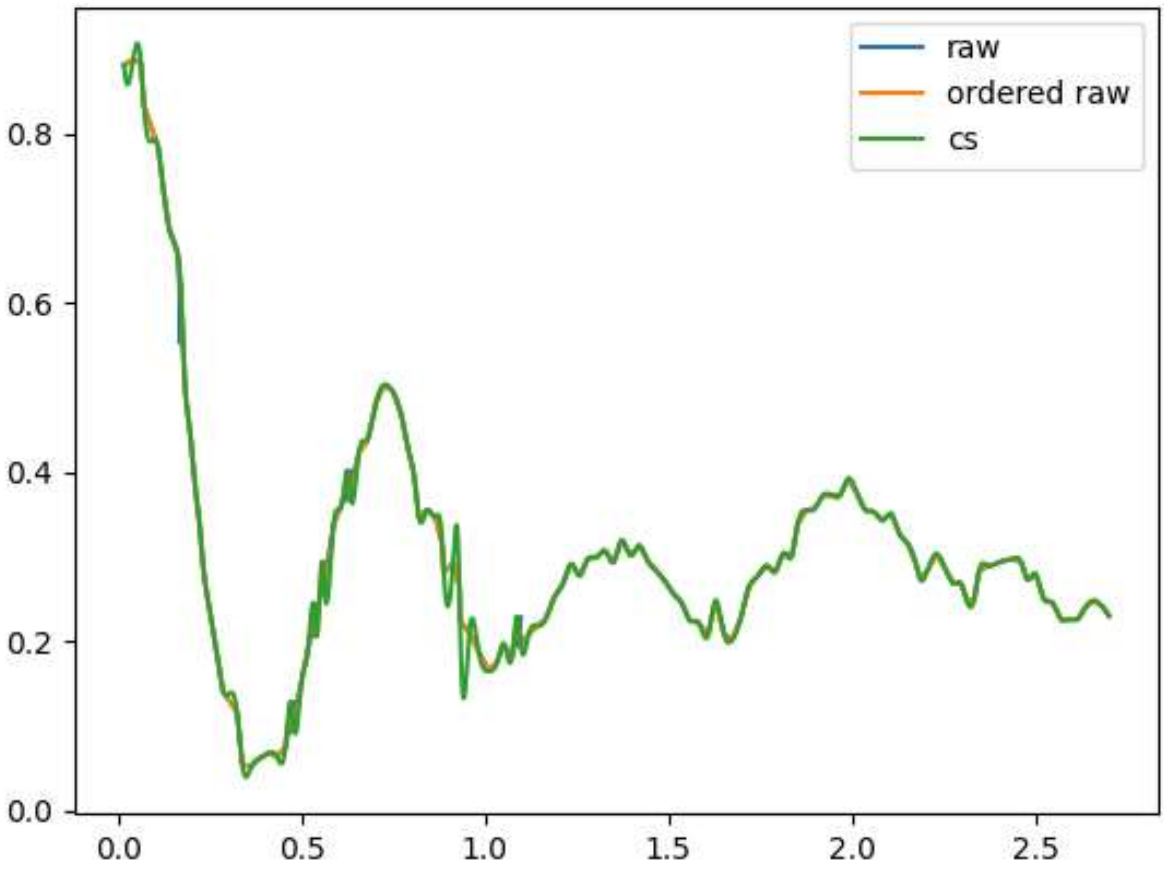
Once the data was ordered a plot was made of both raw scraped data and ordered, see below:



The curves were mostly matching, with some points slightly rearranged so that there is no points going backwards in time:



The sort command I used arranges the array such that all the array values are in ascending order. This however doesn't take into account repeating values in the array, and so I needed to add a loop that found the repeating values and removed the duplicates. With the duplicates removed I could run the cubic spline and got a figure that quite closely matched the raw scraped data:



This was done for all six of the scraped data sets from Fig. 3c from:
'Application of spherical harmonics for DEER data analysis in systems with a conformational

distribution' by Alexey Potapov 2020

See below the script that contains the sorting, cubic splining, and plotting of the scraped data from figure 3c. (Note, the arrays are very large and are not included in this copy and paste of the script)

```
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt
from scipy import interpolate
import numpy as np
import collections
fig, ax1 = plt.subplots()

x = [...] #Black Curve time vals
y = [...] #Black Curve amp vals

x = [...] #red Curve time vals
y = [...] #red Curve amp vals

x = [...] #green Curve time vals
y = [...] #green Curve amp vals

x = [...] #blue Curve time vals
y = [...] #blue Curve amp vals

x = [...] #pink Curve time vals
y = [...] #pink Curve amp vals

x = [...] #yellow Curve time vals
y = [...] #yellow Curve amp vals

ax1.plot(x, y, label='raw')
x.sort()
repeatx = [item for item, count in collections.Counter(x).items() if count > 1] #Finds
the index at which a duplicate is initially found
while len(repeatx) > 0:
    ind = x.index(repeatx[0])

    y.remove(y[ind])
    x.remove(x[ind])
    repeatx = [item for item, count in collections.Counter(x).items() if count > 1]
#Loop repeats until there are no more duplicates

ax1.plot(x, y, label='ordered raw')

cs = CubicSpline(x, y) #Performing cubic spline
xs = np.arange(min(x), max(x), 0.001)
ax1.plot(xs, cs(xs), label='cs')

ax1.legend()
plt.show()
```

Cubic Spline 2

23 February 2021 11:32

Aims:

- To improve the cubic spline script.
- Produce data for all 6 curves from figure 3c, with all corresponding time stamps.

Outcome:
The script was altered slightly and curves were produced from data scraped from all 6 curves of fig. 3c from:
'Application of spherical harmonics for DEER data analysis in systems with a conformational distribution' by Alexey Potapov 2020.

The major change to the script was re-defining the min and max time values so that all vectors are the same length. This script now outputs the cubic spline curves for the 6 curves and the arrays are stored in the console.

Sample of the script updated to correct the different array lengths.

```
tmin = 0
tmax = 2.7

ax1.plot(kt, ka, label='raw')
kt.sort()
repeatx = [item for item, count in collections.Counter(kt).items() if count > 1]
#Finds the index at which a duplicate is initially found
while len(repeatx) > 0:
    ind = kt.index(repeatx[0])

    ka.remove(ka[ind])
    kt.remove(kt[ind])
    repeatx = [item for item, count in collections.Counter(kt).items() if count > 1]
#Loop repeats until there are no more duplicates
```

Data formatting

26 February 2021 20:27

Aim:

- To modify the cubic spline script to print the data to a file

Outcome:

The script has been modified such that it outputs a text document containing the time values on the leftmost column, and the corresponding amplitude values for the different graphs in the following columns to the right.

Some minor modifications and optimisations were made to the code toward the goal of changing the script to work with as many graphs as is required that will be pulled from the research papers, as opposed to being hardcoded to only work with these 6 test graphs.

The string chosen to occupy the space between values was three spaces.

```
0.0   0.6978160449458142   0.7539873087067421   0.9727863269296998   0.9916150779494817   1.0808610066092035   1.1005762882698917
0.001   0.7301207666377912   0.7634838734239874   0.9728197691562066   0.9877958696671488   1.0506241794950706   1.0795372533551073
0.002   0.7574960889812598   0.7725947956574039   0.9727836971719734   0.9841770389363688   1.02291851063992975   1.0596765276941205
0.003   0.7802641809701645   0.7813290143560616   0.9726804415727408   0.9807527116558905   0.9976492054176826   1.040964133525048
0.004   0.7987472115984499   0.7896954684690305   0.9725123329542481   0.9775170137244632   0.9747214692060241   1.0233700930860057
0.005   0.8132673498600601   0.7977030969453806   0.9722817019122361   0.9744640710408361   0.9540405073801199   1.0068644286151103
0.006   0.8241467647489402   0.8053608387341815   0.9719908790424444   0.9715888095037583   0.935511525315768 3   0.9914171623504775
0.007   0.8317076252590341   0.8126776327845037   0.9716421949406132   0.9688829550119787   0.9190397283887677   0.976998316530224
0.008   0.8362721003842866   0.8196624180454166   0.9712379802024826   0.9663430334642469   0.9045303219749159   0.9635779133924658
0.009000000000000001   0.838162359118642   0.8263241334659905   0.9707805654237928   0.9639623707593115   0.891888511450011   0.9511259751753192
0.01   0.837700570456045 1   0.832671717995295   0.9702722812002836   0.9617350927959221   0.8810195021898515   0.9396125241169005
0.011   0.8352089033904397   0.8387141105824004   0.9697154581276951   0.9596553254728274   0.8718284995702354   0.9290075824553261
0.012   0.8310095269157711   0.8444602501763766   0.9691124268017677   0.9577171946887767   0.8642207089669607   0.919281172428712
0.013000000000000001   0.8254246100259831   0.8499190757262932   0.9684655178182412   0.9559148263425192   0.858101335755825 9   0.9104033162751746
0.014   0.8187763217150208   0.8550995261812204   0.9677770617728555   0.954242346332804   0.8533755853126289   0.90234403623283
0.015   0.8113868309768283   0.8600105404902282   0.9670493892613511   0.9526938805583802   0.849948663013168   0.8950733545397949
0.016   0.8035783068053498   0.8646610576023863   0.9662848308794677   0.951263554917997   0.8477257742332411   0.888561293434185
0.017   0.7956729181945302   0.8690600164667649   0.9654857172229456   0.9499454953104033   0.8466121243486466   0.8827778751541167
0.018000000000000002   0.7879928341383141   0.8732163560324337   0.9646543788875249   0.9487338276343484   0.8465129187351828   0.8776931219377064
0.019   0.7808602236306458   0.8771390152484628   0.9637931464689454   0.9476226777885814   0.8473333627686475   0.8732770560230704
0.02   0.7745972556654694   0.8808369330639221   0.9629043505629475   0.9466061716718515   0.8489786618248389   0.8694996996483246
0.021   0.7695260992367299   0.8843190484278816   0.9619903217652709   0.9456784351829076   0.8513540212795555   0.8663310750515858
0.022   0.7659140959867411   0.8875943002894109   0.9610533906716 56   0.9448335942204992   0.8543646465085951   0.8637412044709697
0.023   0.7637536441433999   0.8906716275975803   0.9600958878778427   0.944065774683375   0.857915742887756   0.861700110144593
0.024   0.7629511399333901   0.8935599693014599   0.9591201439795711   0.9433691024702846   0.8619125157928363   0.8601778143105716
0.025   0.7634129502114421   0.8962682643501191   0.9581284895725813   0.9427377034799767   0.8662601705996344   0.859144339207022
0.026000000000000002   0.7650454418322864   0.8988054516926283   0.9571232552526132   0.9421657036112006   0.8708639126839481   0.8585697070720605
```

Scraping data from newly found papers

04 March 2021 16:04

Aim:

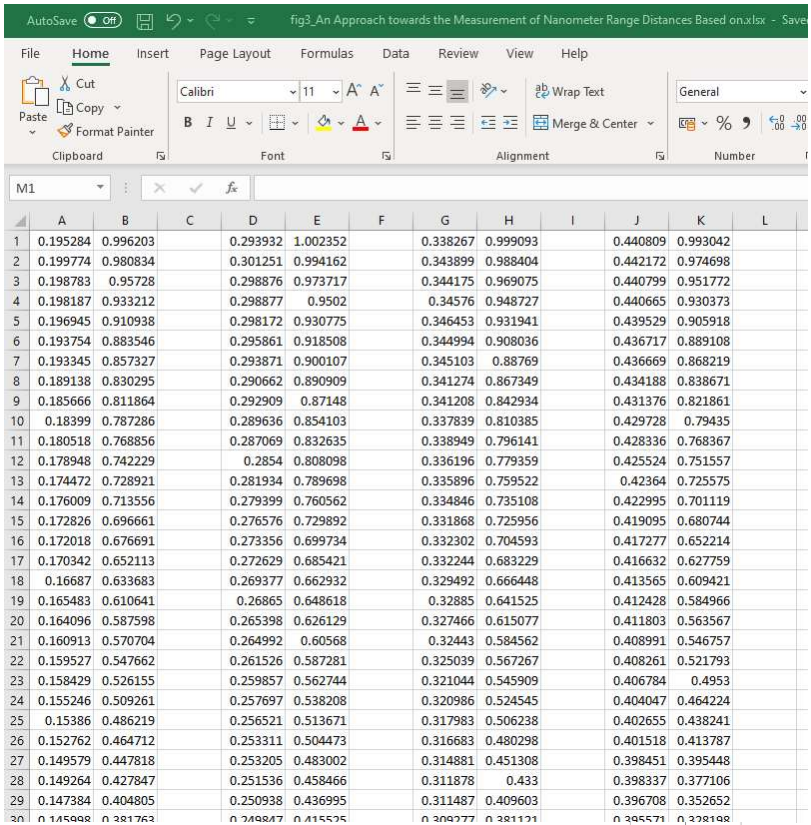
- To scrape data from curves found in new papers gathered.

Outcomes:

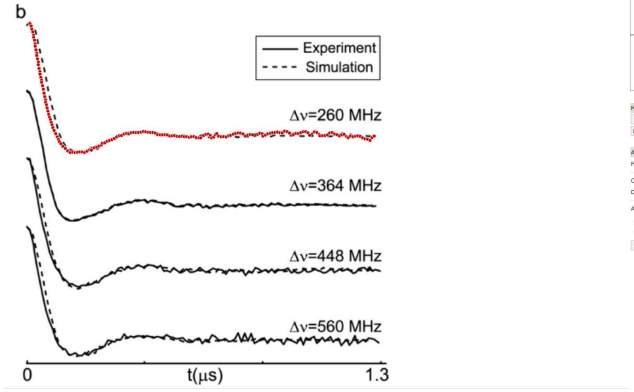
The papers used for scraping are:

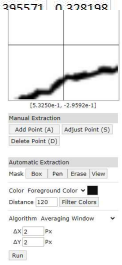
1. ''An Approach towards the Measurement of Nanometer Range Distances Based on Cu2+ Ions and ESR'' By: Zhongyu Yang, Drew Kise, and Sunil Saxena, 2009. Figures 3 and 9 used.
2. ''Direct assignment of EPR spectra to structurally defined iron-sulfur clusters in complex I by double electron–electron resonance'' By: Maxie M. Roessler, et al., 2009. Figure 5 used
3. ''Conformational flexibility of nitroxide biradicals determined by X-band PELDOR experiments'' By: D. Margraf, et al. 2007. Figure 4 used.
4. ''A pulsed EPR method to determine distances between paramagnetic centers with strong spectral anisotropy and radicals: The dead-time free RIDME sequence'' By: Sergey Milikisyants, et al. 2009. Figure 5 used.

Note: when the data is scraped and saved to an excel file, the columns starting from A moving right will be the times in the first column and then the amplitudes in the second. These pairs (starting from A) will correspond to the bottom curve in the figure moving upwards:



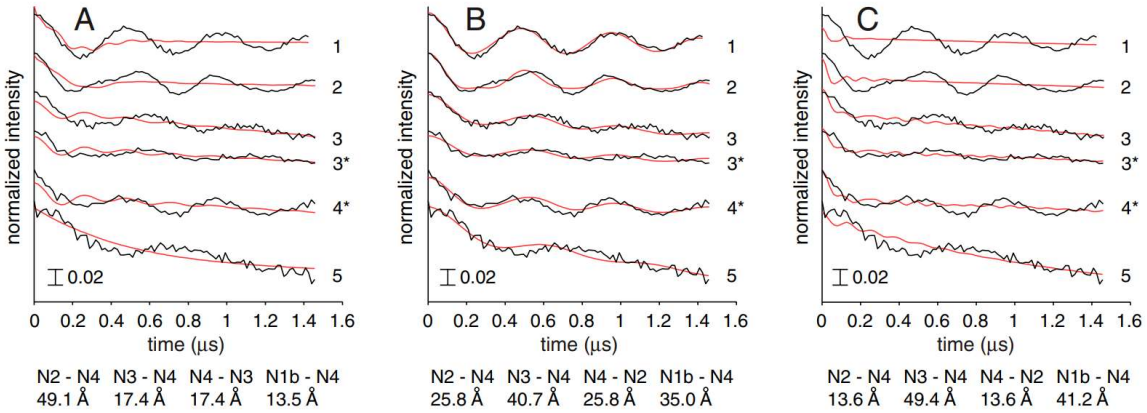
	A	B	C	D	E	F	G	H	I	J	K	L
1	0.195284	0.996203		0.293932	1.002352		0.338267	0.999093		0.440809	0.993042	
2	0.199774	0.980834		0.301251	0.994162		0.343899	0.988404		0.442172	0.974698	
3	0.198783	0.95728		0.298876	0.973717		0.344175	0.969075		0.440799	0.951772	
4	0.198187	0.933212		0.298877	0.9502		0.34576	0.948727		0.440665	0.930373	
5	0.196945	0.910938		0.298172	0.930775		0.346453	0.931941		0.439529	0.905918	
6	0.193754	0.883546		0.295861	0.918508		0.344994	0.908036		0.436717	0.889108	
7	0.193345	0.857327		0.293871	0.900107		0.345103	0.88769		0.436669	0.868219	
8	0.189138	0.830295		0.290662	0.890909		0.341274	0.867349		0.434188	0.838671	
9	0.185666	0.811864		0.292909	0.87148		0.341208	0.842934		0.431376	0.821861	
10	0.18399	0.787286		0.289636	0.854103		0.337839	0.810385		0.429728	0.79435	
11	0.180518	0.768856		0.287069	0.832635		0.338949	0.796141		0.428336	0.768367	
12	0.178948	0.742229		0.2854	0.808098		0.336196	0.779359		0.425524	0.751557	
13	0.174472	0.728921		0.281934	0.789698		0.335896	0.759522		0.42364	0.725575	
14	0.176009	0.713556		0.279399	0.760562		0.334846	0.735108		0.422995	0.701119	
15	0.172826	0.696661		0.276576	0.729892		0.331868	0.725956		0.419095	0.680744	
16	0.172018	0.676691		0.273356	0.699734		0.332302	0.704593		0.417277	0.652214	
17	0.170342	0.652113		0.272629	0.685421		0.332244	0.683229		0.416632	0.627759	
18	0.16687	0.633683		0.269377	0.662932		0.329492	0.666448		0.413565	0.609421	
19	0.165483	0.610641		0.26865	0.648618		0.32885	0.641525		0.412428	0.584966	
20	0.164096	0.587598		0.265398	0.626129		0.327466	0.615077		0.411803	0.563567	
21	0.160913	0.570704		0.264992	0.60568		0.32443	0.584562		0.408991	0.546757	
22	0.159527	0.547662		0.261526	0.587281		0.325039	0.567267		0.408261	0.521793	
23	0.158429	0.526155		0.259857	0.562744		0.321044	0.545909		0.406784	0.4953	
24	0.155246	0.509261		0.257697	0.538208		0.320986	0.524545		0.404047	0.464224	
25	0.15386	0.486219		0.256521	0.513671		0.317983	0.506238		0.402655	0.438241	
26	0.152762	0.464712		0.253311	0.504473		0.316683	0.480298		0.401518	0.413787	
27	0.149579	0.447818		0.253205	0.483002		0.314881	0.451308		0.398451	0.395448	
28	0.149264	0.427847		0.251536	0.458466		0.311878	0.433		0.398337	0.377106	
29	0.147384	0.404805		0.250938	0.436995		0.311487	0.409603		0.396708	0.352652	
30	0.145998	0.381763		0.249847	0.415525		0.309277	0.381121		0.395571	0.328198	



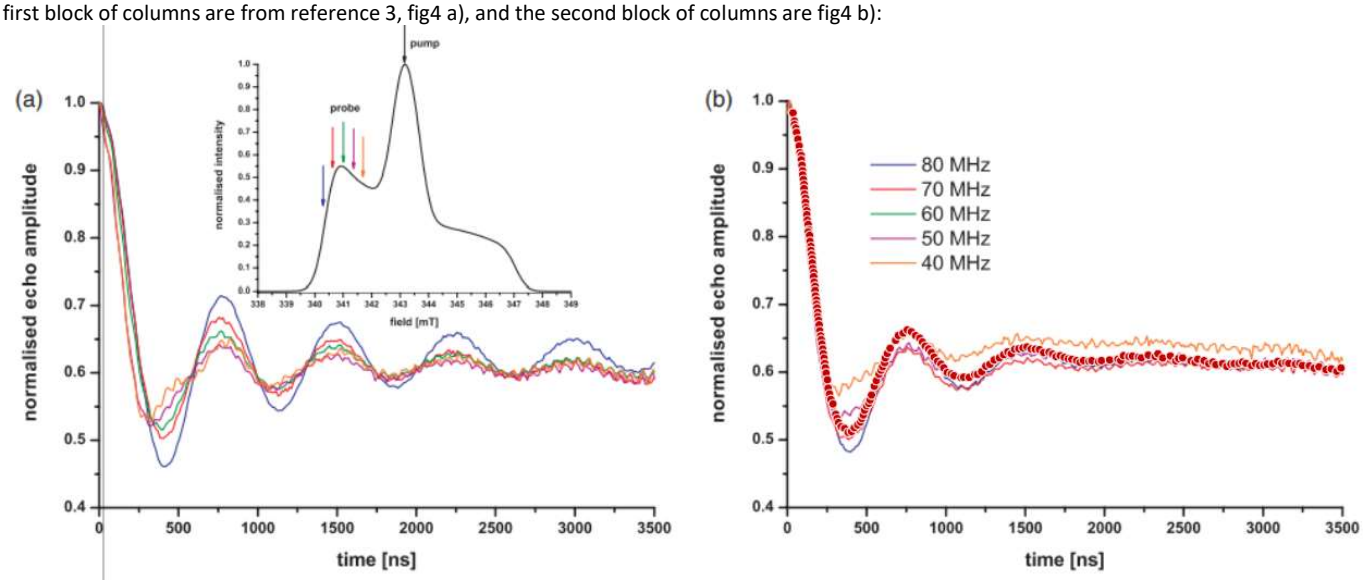


So for the above images, columns A(time) and B(amplitude) correspond to the bottom curve, columns D and E correspond to the second curve from the bottom, etc.

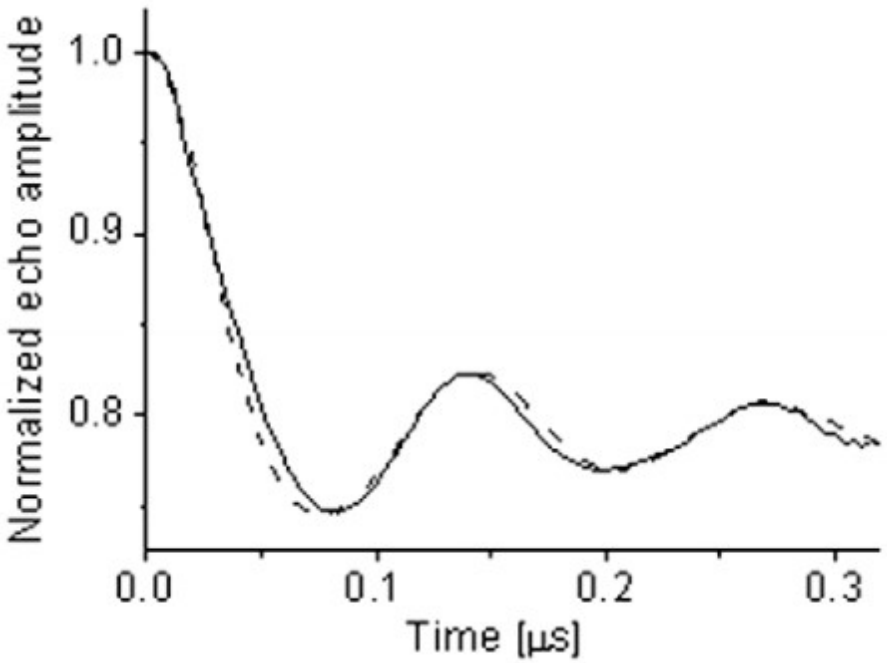
For the paper from reference 2, the order of storing data is the same as described above starting from fig.5 A, then the next block of columns from fig.5 B, etc.



The first block of columns are from reference 3, fig4 a), and the second block of columns are fig4 b):



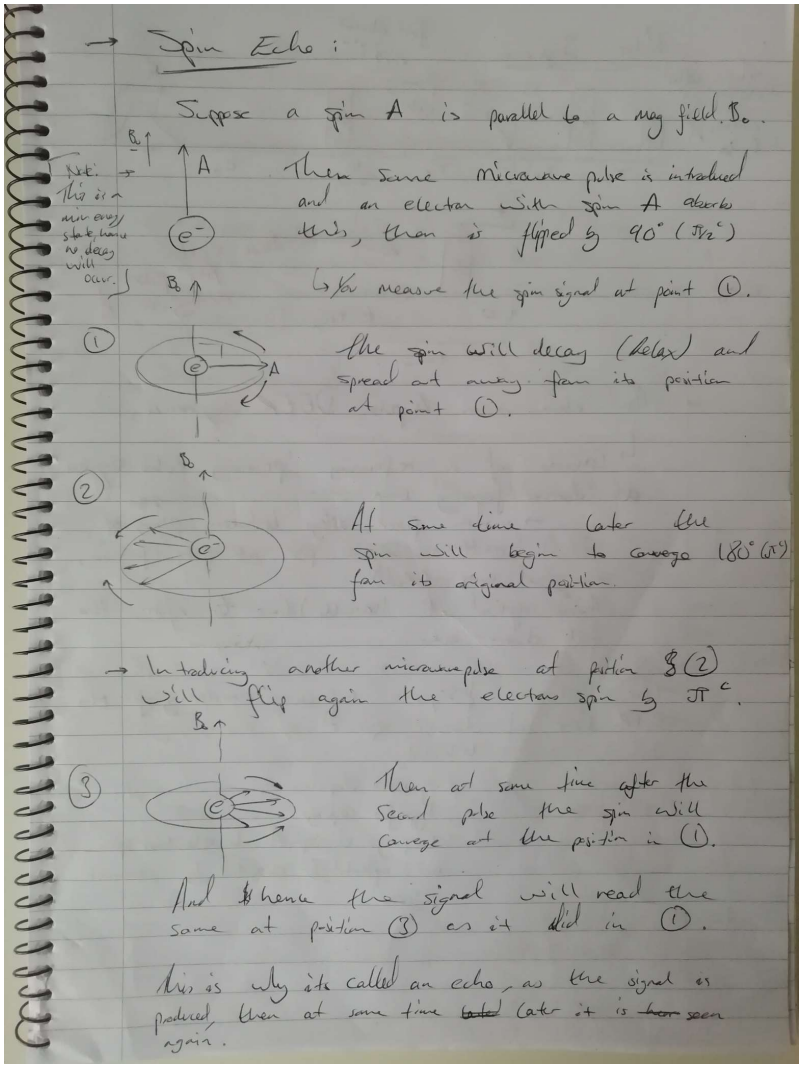
Reference 4 fig 5:

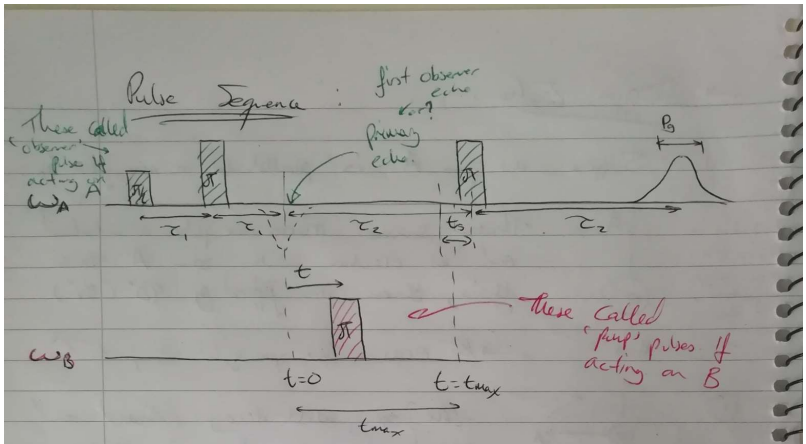

Normalized echo amplitude
1.0
0.9
0.8
0.0
0.1
0.2
0.3
Time [μs]

Background theory to date

04 March 2021 16:15

As the weeks have progressed we have been reading through the background theory that underlines this project, to date this is what we have:





→ The above is a 4 pulse DEER sequence.

↳ Consists of a refocusing 'primary echo sequence' at observer frequency ω_A.

→ with fixed delays between pulses, and an inversion pulse at 'pump' frequency ω_B (Red).

These applied at (Variable) time t after the first observer echo.

The observer sequence refocuses broadening of the EPR line of observer spin A. (inhomogeneous)

↳ Including:

- g value dispersion.
- Hyperfine coupling.
- Coupling of spin A to electron spins not ~~affected by~~ excited by observer pulses

Transverse relaxation:

↳ Loss of phase coherence of electron spin magnetization and hence loss of signal due to random mag. field fluctuations.

- Transverse relaxation of spin A and coupling to other electron spins excited by the observer pulses lead to loss of echo intensity (attenuation) by a factor of $\rightarrow \exp[-2k(\tau_1 + \tau_2)]$

where k = decay const., $k = \frac{1}{T_{2,A}} + k_{ID}$

$T_{2,A}$ = Transverse relaxation time

k_{ID} = Instantaneous diffusion rate:

$k_{ID} = C_A K_A$, C_A = Concentration of A spins

K_A = Instantaneous diffusion strength (proportional to intensity of observer π pulses)

~~Note: for Nitroxide spin labels at X-Band frequencies (~9.6 GHz)~~

from the spins B that are coupled to spin A, only a fraction are excited by the pump pulse frequency ω_B. This fraction is $\lambda < 1$. So only this fraction have their spin states inverted.

⇒ With a 12 ns pump pulse at X-band freq ≈ 9.6 GHz the inversion efficiency for nitroxide labels is $\lambda \approx 0.5$.

→ Inversion of spin B_i changes the frequency of spin A_z by the electron-electron coupling freq $\omega_{ee,i}$ (or width), this leads to a phase gain of:

$$\phi_i = \omega_{ee,i} t$$

[Note: this phase change only affects the fraction λ_i of the spin A magnetization]

→ the echo amplitude is given as:

$$V(t) = \prod_i \{1 - \lambda_i (1 - \cos(\omega_{ee,i} t))\} \quad (1)$$

Running over all B_i spins coupled to spin A.

[Note: the echo observed is always that of spin A, this is likely because A is the reference, and hence its position is known.]

→ If excitation Bandwidths (both observer and pump) are much larger than the $e^- - e^-$ coupling then λ_i from eq. 1 are independent of $e^- - e^-$ coupling.

Ask Stefan.

↳ If the pump pulse is used this will affect the phase of the observer echo.
→ this shift is "fixed" by a Block-Selgert shift and is corrected for by receiver phase adjustment.

Using DEER to measure distances requires the following assumptions:

Assumpt 1/.

- Exchange coupling between ~~electrons~~ electron spins is neglected.
- Both spins are assumed to be quantized along the external mag field.

With these assumptions the coupling simplifies to the mag. dipole coupling where:

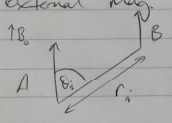
$$\omega_{dd,ii} = \left(\frac{C_i}{r_i^3}\right) \cdot \left(1 - 3\cos^2\theta_i\right).$$

↳ where C_i is proportional to the g-values of the A and B spins.

And for: $g_A = g_B = 2.0055$ (the isotopic g val. for nitroxide labels)

$$C_i = 52.2\ \text{MHz nm}^3.$$

θ_i = the angle between the spin-spin vector and the external mag field.



→ Note: Exchange Coupling!

↳ Interaction between electron spins due to overlapping wave functions.

→ this decays exponentially with distance.

Assump. 2/ - A semi-isolated spin pair is assumed.

↳ meaning: for a given/known spin A only one spin B is within the ideal (sensitive) range for DEER.

- All other Bi spins in other molecules are assumed to be Bo homogeneously distributed in space.

example.
Note: for membrane proteins and in liposomes the homogeneous spacial distribution can be reduced to a 2-D homogeneous spacial distribution ↓.

fractional
This can be relaxed to a ^D dimensional spacial homogeneous distribution in general only!

Assump. 3/ - the correlation between λ_i and $\omega_{dd,i}$, specifically from θ_i (the link between λ and ω_{dd} that comes from each depending on θ_i) is ignored (neglected) and an average orientation is taken.

Using these Assumptions equation 1 ~~now~~ converts to an expression for a macroscopically disordered sample:

$$I = \int_0^1 \cos(\omega_{dd_i} t) d\cos\theta$$

$$V(t) = \left\{ 1 - \lambda \left[1 - \int_0^1 \cos\left(\frac{C_i}{r_i^3} (1 - 3\cos^2\theta_i) t \right) d\cos\theta \right] \right\} B(t) \quad (2)$$

- $B(t)$ is the ~~eff~~ "background" function and is here due to all of the other possible electric charges in the molecule ~~acting~~ acting on A (or B I guess?)

↳ ~~the~~ And $B(t)$ takes the form:

$$B(t) = \exp\left(-C_B K_B t^{D/3}\right).$$

↳ C_B = Concentration of B spins
K_B = Instantaneous diffusion strength
D = Dimensions of the homogeneous spatial Distribution.

- If more than ~~one~~ one B spin is within the sensitive range for DEER then eq. 2 takes the form:

$$V(t) = F(t) B(t)$$

where $F(t)$ ← called the form factor (likely due to its containing of r_i and θ_i information).

And $F(t)$ is the product of all possible pair contributions.

Note: An analytical expression for $B(t)$ is not known, and the form depends on the excitation profiles of the pulses and the shape of the EPR lines.

Cubic Spline Code – General

09 March 2021 00:46

Aim:

- To modify the cubic spline script to accept many inputs, as it currently is hard coded to only do the 6 test graphs. The intention is to make it so the code can accept any number of new graphs, with any numbers of points in each graph, and output this formatted, cubic-splined data.

Outcome:

The code has been modified such that it now accepts inputs as required, with the code being generalised. Two functions were added: one to initialise the data and account for the fact the data comes with differing quantities of time and amplitude values per graph, and one to build the print string.

In order to input data using this script, the following format must be used:

```
kt = [0.017222525,
ka = [0.88158701,

initialise_data(kt, ka, 1)

rt = [0.009517343,
ra = [0.886050158,

initialise_data(rt, ra, 2)

gt = [0.009517343,
ga = [0.962791325,

initialise_data(gt, ga, 3)

bt = [0.009517343,
ba = [0.970523812,

initialise_data(bt, ba, 4)

pt = [0.013369934,
pa = [0.851867467,

initialise_data(pt, pa, 5)

yt = [0.001812162,
ya = [0.752715121,

initialise_data(yt, ya, 6)
```

Whereby the data is pasted into variable names defined by the user, and input into the initialisation function as shown in the image. The third parameter of the initialisation function must be given as the number of that graph.

It is worth noting that this is not the most optimised method of inputting, formatting (using cubic spline) and outputting this data. In future, this script will be modified again to instead read data from the text file directly output from WebPlotDigitiser. The reason the code has been done this way temporarily is due to the fact the most recently scraped data has instead been stored in an Excel document, and so the script has been currently set up such that the data can be copied and pasted directly from Excel.

Background theory (SVD)

13 March 2021 14:55

Single valued decomposition will be used as part of the sifting algorithm that we will use on the scraped data, here are the notes I have made so far on the topic to aid our understanding:

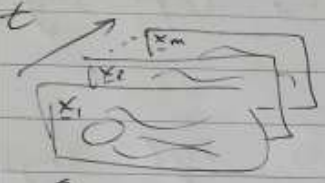
Singular Valued Decomposition (SVD):

Space save matrix $\underline{\underline{X}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix}$

where each column is a vector $\underline{x}_1, \underline{x}_2, \cdots, \underline{x}_m$.

where $x_k \in \mathbb{R}^n$ ← n is the length of vector.

→ Same example:
You have a fluid flow evolving with time:



Each snapshot contains information stored in the x_k vectors

→ the $\underline{\underline{X}}$ matrix is ~~the~~ describing the state of the system as it evolves over time.

We can re-write matrix $\underline{\underline{X}}$ as product of 3 other matrices $\underline{\underline{U}}\ \underline{\underline{\Sigma}}\ \underline{\underline{V}}$, by use of SVD:

$$\therefore \underline{\underline{X}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} = \underline{\underline{U}}\ \underline{\underline{\Sigma}}\ \underline{\underline{V}}^T$$

$\underline{\underline{U}}$ and $\underline{\underline{V}}$ are Unitary Matrices
$\underline{\underline{\Sigma}}$ is a diagonal Matrix.

$$\hookrightarrow \underline{\underline{U}}\ \underline{\underline{\Sigma}}\ \underline{\underline{V}}^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \sigma_m \\ 0 & & & \end{bmatrix} \underbrace{\begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_m \\ | & | & & | \end{bmatrix}^T}_{m \times m \text{ Square}}$$

$n \times n$ Square

Same shape as $\underline{x}_k$, Ordered hierarchically e.g. the u_1 is more important than u_2, etc.

Think of them as eigen-symbols of fluid flow as described above

→ terms in order of their ability to describe the variance in the Columns of $\underline{\underline{X}}$

U and V are unitary:

$U^T U = U U^T = \mathbb{1}_{n \times n}$ $\quad V^T V = V V^T = \mathbb{1}_{m \times m}$

↳ $U^T \equiv U^{-1}$, $V^T \equiv V^{-1}$

Σ is diagonal, non-negative and ordered hierarchally

↳ $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \sigma_m \geq 0$.

→ Calls the U matrix the left singular matrix

→ the V matrix the right singular matrix

→ the Σ matrix a matrix of singular values.

Using this set up we will be able to "cut off" the smaller σ values that are least dominant and hence reduce and approximate the matrix X only in terms of the most dominant features.

$$\Rightarrow \underline{\underline{X}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} = \underline{\underline{U}}\,\underline{\underline{\Sigma}}\,\underline{\underline{V}}^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_m \\ \hline 0 & & & \end{bmatrix} \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_m \\ | & | & & | \end{bmatrix}^T$$

ordered by importance

↑ And hence the columns of $\underline{\underline{U}}$ and $\underline{\underline{V}}^T$ also are ordered by "importance".

Understanding the matrices:

$\underline{\underline{U}}$ has columns of equivalent size to the columns of $\underline{\underline{X}}$ and so for our example the columns of $\underline{\underline{U}}$ can be reshaped into the "eigen" flow stories snap shots.

$\underline{\underline{\Sigma}}$ are heirarchally organised to show relative importance of each column with respect to the original data set.

$\underline{\underline{V}}$:

In the context of the fluid flow each snap shot will contain some amount of the previous snap shot.
e.g: x_2 will contain some fraction of x_1 and so on.

So the columns of $\underline{\underline{V}}^T$ are like the "eigen" time series of the system.

→ How much the "eig. columns" carry over to later columns ? ← Check this.

The columns of $\underline{\underline{V}}^T$ will give the "mixture of all $\underline{\underline{U}}$ ~~that make~~ that make up each corresponding column of $\underline{\underline{X}}$

All scaled by the singular value.

e.g: column 1 of $\underline{\underline{V}}^T$ will be the mix of $\underline{\underline{U}}$ columns that'll make up column 1 of $\underline{\underline{X}}$

$$\underline{\underline{X}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} = \underline{\underline{U}}\,\underline{\underline{\Sigma}}\,\underline{\underline{V}}^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ 0 & & & \sigma_m \end{bmatrix} \begin{bmatrix} - & v_1^T & - \\ - & v_2^T & - \\ & \vdots & \end{bmatrix}$$

for n cols of $\underline{\underline{X}}$, only the first m cols of $\underline{\underline{U}}$ are important.

→ Multiplying these matrices out will give:

$$\underline{\underline{X}} = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + \sigma_m u_m v_m^T + 0$$

(zero for all n terms greater than m.)

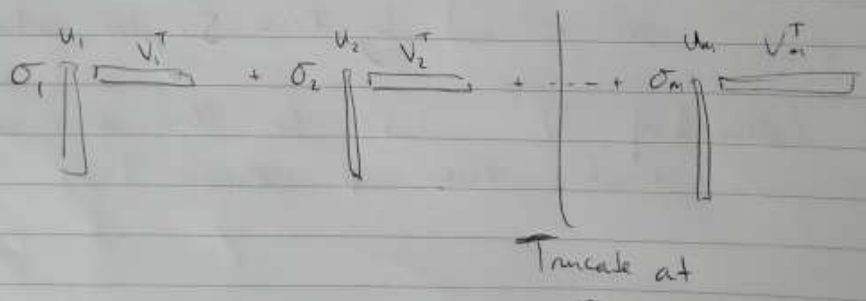
→ We can call these 'reduced' matrices:

$$\underline{\underline{X}} = \hat{\underline{\underline{U}}}\,\hat{\underline{\underline{\Sigma}}}\,\underline{\underline{V}}^T \quad (= \underline{\underline{U}}\,\underline{\underline{\Sigma}}\,\underline{\underline{V}}^T).$$

"economy SVD".

Note: During this set up, the assumption that $n \gg m$.

$\sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \cdots + \sigma_m u_m v_m^T =$



Truncate at r

A truncation often is introduced to correspond to increasingly small σ_i values.

↳ If the $\sigma_i \gg \sigma_j$ for example, including σ_j will be of minimal benefit to the approximated model.

→ After taking the truncation at r (r-rank approx.)

$$\underline{\underline{X}} \simeq \underline{\underline{\tilde{U}}}\,\underline{\underline{\tilde{\Sigma}}}\,\underline{\underline{\tilde{V}}}^T$$

[look into: Eckard Young Theorem]

Now due to the truncation truncation $\underline{\underline{\tilde{U}}}$ and $\underline{\underline{\tilde{V}}}^T$ are not unitary:

$$\therefore \underline{\underline{\tilde{U}}}^T\underline{\underline{\tilde{U}}} = \mathbb{1}_{rr}, \text{ but } \underline{\underline{\tilde{U}}}\,\underline{\underline{\tilde{U}}} \neq \mathbb{1}.$$

$$\underline{\underline{X}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} = \underline{\underline{U}}\,\underline{\underline{\Sigma}}\,\underline{\underline{V}}^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \sigma_m \\ 0 & & & \end{bmatrix} \begin{bmatrix} - & v_1^T & - \\ - & v_2^T & - \\ & \vdots & \end{bmatrix}$$

$$\approx \hat{\underline{\underline{U}}}\,\hat{\underline{\underline{\Sigma}}}\,\underline{\underline{V}}^T$$

"Economy SVD"

Suppose you take

$\underline{\underline{X}}^T\underline{\underline{X}}$ ($m \times m$) $=$ ($m \times n$) $\cdot$ ($n \times m$)

$$= \begin{bmatrix} x_1^T x_1 & x_1^T x_2 & \cdots & x_1^T x_m \\ x_2^T x_1 & x_2^T x_2 & \cdots & x_2^T x_m \\ \vdots & \vdots & & \vdots \\ x_m^T x_1 & x_m^T x_2 & \cdots & x_m^T x_m \end{bmatrix}$$

Each element, $x_i^T x_j = \langle x_i, x_j \rangle$ ← Inner product.

Correlation Matrix } Compute U, Σ, V from these correlation matrices as eig. vects. and eig. vals of these.

↳ ~~If $\underline{\underline{X}} = \underline{\underline{U}}\,\underline{\underline{\Sigma}}\,\underline{\underline{V}}^T$~~ If $\underline{\underline{X}} = \hat{\underline{\underline{U}}}\,\hat{\underline{\underline{\Sigma}}}\,\underline{\underline{V}}^T$, then $\underline{\underline{X}}^T = \underline{\underline{V}}\,\hat{\underline{\underline{\Sigma}}}\,\hat{\underline{\underline{U}}}^T$

$$\therefore \underline{\underline{X}}^T\underline{\underline{X}} = \underline{\underline{V}}\,\hat{\underline{\underline{\Sigma}}}\,\underbrace{\hat{\underline{\underline{U}}}^T\hat{\underline{\underline{U}}}}_{=\mathbb{1}}\,\hat{\underline{\underline{\Sigma}}}\,\underline{\underline{V}}^T = \underline{\underline{V}}\,\hat{\underline{\underline{\Sigma}}}^2\,\underline{\underline{V}}^T$$

[Note: this isn't how $U\Sigma V^T$ are calculated though it is a nice interpretation]

↳ $\underline{\underline{X}}^T\underline{\underline{X}}\,\underline{\underline{V}} = \underline{\underline{V}}\,\hat{\underline{\underline{\Sigma}}}^2$

eig. vectors, eig. vals of correlation matrix $\underline{\underline{X}}^T\underline{\underline{X}}$

Cubic Spline Code – File Reading

15 March 2021 22:10

Aims:

- To modify the cubic spline script so that it accepts an input of a text file in the format directly output by WebPlotDigitiser
- To modify the cubic spline script so that it lists the columns on the output text file

Outcome:

The cubic spline script has been modified to theoretically complete both of the aims. This modified script has, however, yet to be tested as more data must be scraped from more papers in text file format to test this new script works. Old data could be re-scraped, but in the interest of time the tests will be run with new data.

```
def split_string(str_i, sep, n):
    sep_ind = str_i.find(sep)
    if n = 1:
        return str_i[0:sep_ind]
    else:
        return str_i[sep_ind, len(str_i) - 1]



t_m = np.zeros((g_count, pt_max))
a_m = np.zeros((g_count, pt_max))

for i in range(0, g_count):
    readData_t = []
    readData_a = []
    fileStr = 'deer_' + str(i)
    fileRead = open(fileStr, 'r').readlines()
    for n in range(0, len(fileRead)):
        readData_t.append(int(split_string(fileRead[n], separationstr_r, 1)))
        readData_a.append(int(split_string(fileRead[n], separationstr_r, 2)))

    initialise_data(readData_t, readData_a, i)
```

Code added to read the data from the text file directly.

Testing of Filereading Code

22 March 2021 17:38

The code modified in the 15th of March entry has been tested and works correctly. Part of the code has been modified to accept custom amplitude labels, with the default being "A_X" where X is the corresponding graph.

This testing and slight modification was performed on Tuesday 16th March 2021.
The code has been placed into the bottom of this entry.

```
from scipy.interpolate import CubicSpline
import numpy as np
import collections
import os.path as op

# Number of graphs
g_count = 2
# Optional Curve labels - leave blank for automatic labelling
labels = ['Curve 1', 'Curve 2']



# String to separate numerical values
separationstr_wr = "   "
separationstr_r = ", "



if len(labels) != g_count:
    autolabels = []
    for i in range(0, g_count):
        autolabels.append("A_" + str(i+1))
else:
    autolabels = labels

def initialise_data(t, y, n):
    global t_m
    global a_m

    n -= 1

    for i in range(0, len(t)):
        t_m[n, i] = t[i]
```

```
        a_m[n, i] = y[i]

    if len(t) < pt_max:
        for i in range(len(t), pt_max):
            t_m[n, i] = np.NaN
            a_m[n, i] = np.NaN

def file_len(fname):
    with open(fname) as f:
        for i, l in enumerate(f):
            pass
    return i + 1



def build_wr_str(t, a, n):
    global g_count
    global separationstr_wr

    wr_str = str(t[n]) + separationstr_wr

    for i in range(0, g_count):
        wr_str = wr_str + str(a[i])
        if i != g_count-1:
            wr_str = wr_str + separationstr_wr
        else:
            wr_str = wr_str + "\n"

    return wr_str

def split_string(str_i, sep, n):
    sep_ind = str_i.find(sep)
    if n == 1:
        return str_i[0:sep_ind]
    else:
        return str_i[(sep_ind + len(sep)):(len(str_i) - 1)]

tmin = 0
tmax = 0
pt_max = 0

for i in range(0, g_count):
```

```
    fileStr = 'deer_' + str(i+1) + '.txt'
    count = file_len(fileStr)

    if count > pt_max:
        pt_max = count

t_m = np.zeros((g_count, pt_max))
a_m = np.zeros((g_count, pt_max))


for i in range(0, g_count):
    readData_t = []
    readData_a = []
    fileStr = 'deer_' + str(i+1) + '.txt'
    fileRead = open(fileStr, 'r').readlines()
    for n in range(0, len(fileRead)):
        readData_t.append(float(split_string(fileRead[n], separationstr_r, 1)))
        readData_a.append(float(split_string(fileRead[n], separationstr_r, 2)))

    if max(readData_t) > tmax:
        tmax = max(readData_t)

    initialise_data(readData_t, readData_a, i)



ts = np.arange(tmin, tmax, 0.001)

splines = []

for n in range(0, g_count):
    current_set_t = t_m[n,:].tolist()
    current_set_a = a_m[n,:].tolist()

    for i in range(pt_max-1, 0, -1):
        if np.isnan(current_set_t[i]):
            del current_set_t[i]
            del current_set_a[i]

    current_set_t.sort()
```

```
   repeatx = [item for item, count in collections.Counter(current_set_t).items() if count > 1] #Finds
the index at which a duplicate is initially found
   while len(repeatx) > 0:
      ind = current_set_t.index(repeatx[0])


      current_set_a.remove(current_set_a[ind])
      current_set_t.remove(current_set_t[ind])
      repeatx = [item for item, count in collections.Counter(current_set_t).items() if count > 1]
#Loop repeats until there are no more duplicates


   splines.append(CubicSpline(current_set_t, current_set_a)) #Performing cubic spline



# Filewriting Part


fy = open("deer_output.txt", "a")
sp_af = np.zeros((g_count, int(tmax*1000)+1))

namestr = "TIME:" + separationstr_wr


for n in range(0, g_count):
   current_af = splines[n](ts)
   for i in range(0, len(ts)):
      sp_af[n,i] = current_af[i]


   if n == g_count -1:
      namestr = namestr + autolabels[n] + "\n"
   else:
      namestr = namestr + autolabels[n] + separationstr_wr


fy.write(namestr)


for i in range(0, len(ts)):
   print_vector = []
   for n in range(0, g_count):
      print_vector.append(sp_af[n,i])


   fy.write(build_wr_str(ts, print_vector, i))
fy.close()
```

Formatting data and updating sifting algorithm

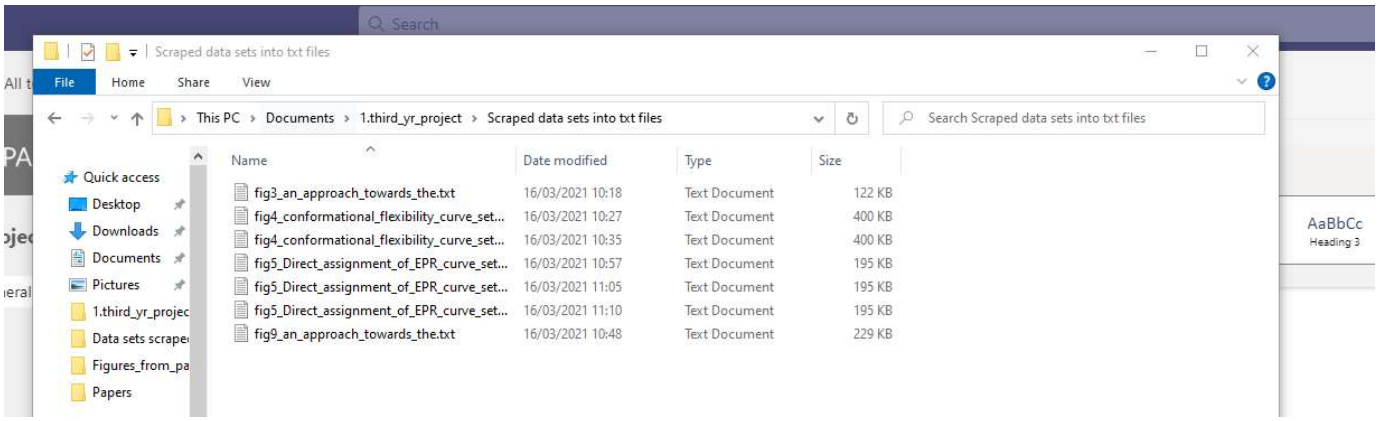
18 March 2021 20:59

Aims:

- To convert the excel files containing the scraped data into .txt files using the code described in the previous entry.
- Alter the sifting algorithm supplied by Dr. Potapov to perform SVD on the data sets scraped from various papers.

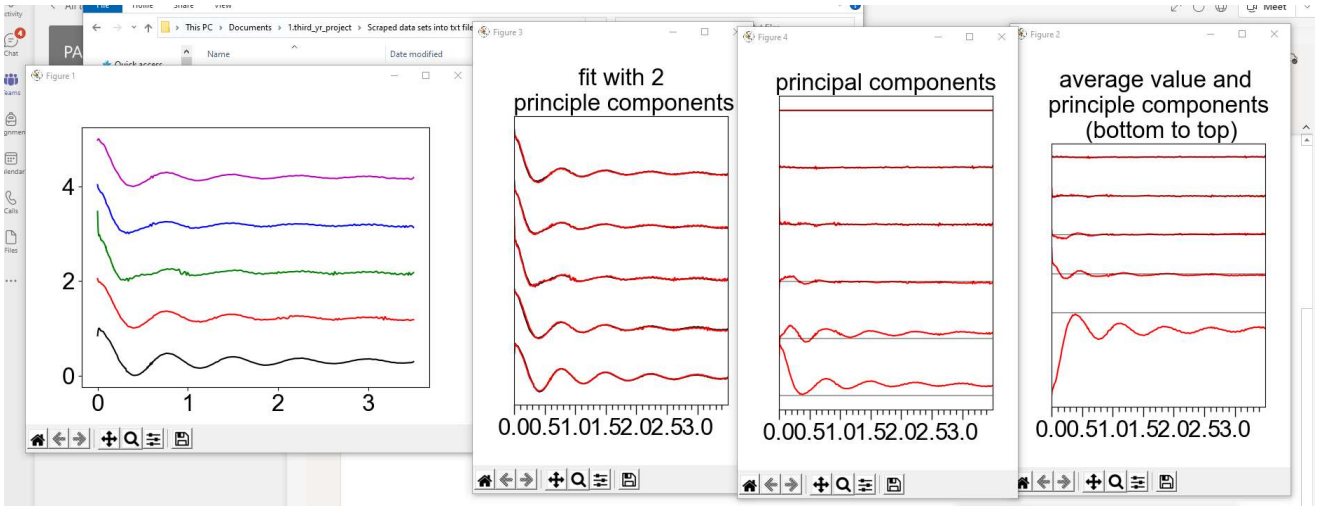
Outcome:

The data contained in the excel files scraped from the papers referenced in the March 4th entry was ran through the script to output the .txt files formatted for the sifting algorithms.

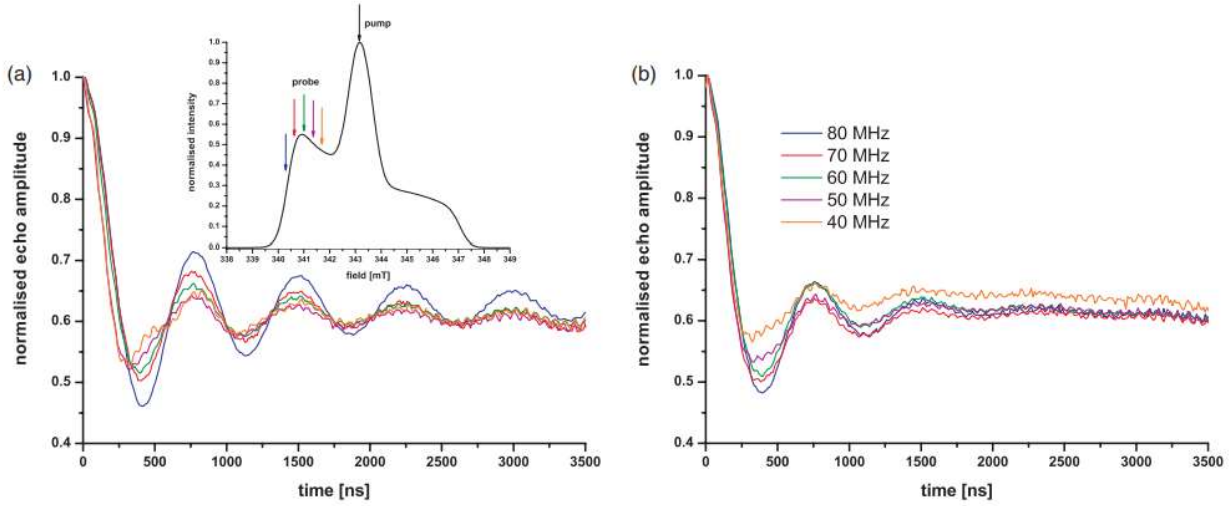


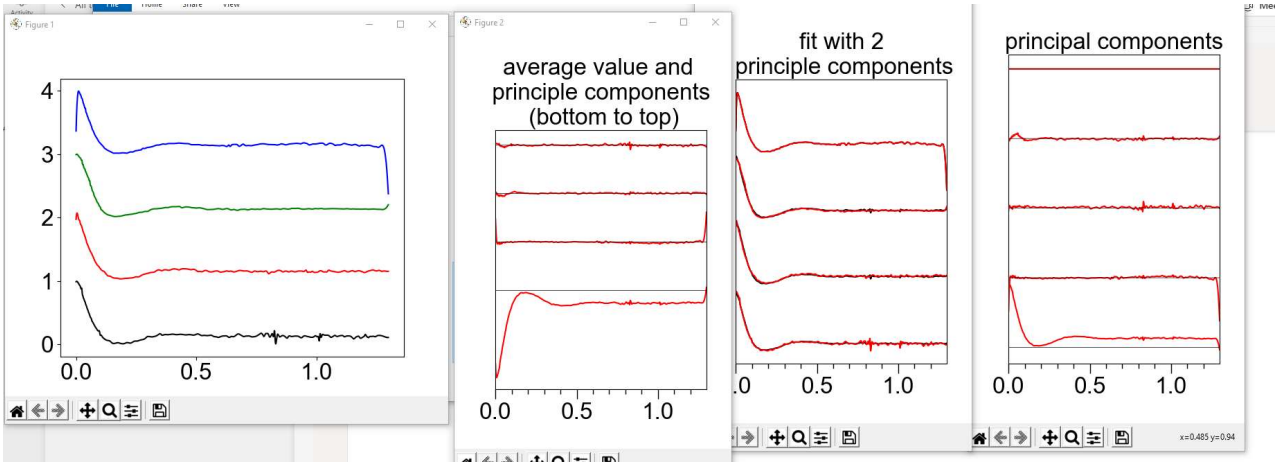
Each of these files containing data for 4-6 DEER traces.

The code (see below) plotted the scraped data alongside the principle component curves calculated using the inbuilt SVD function. Also a plot of the combination of the PC curves matched to the true curves were plotted, showing promising results as a small number of the principle components were needed to rebuild the original curve up to reasonable accuracy.

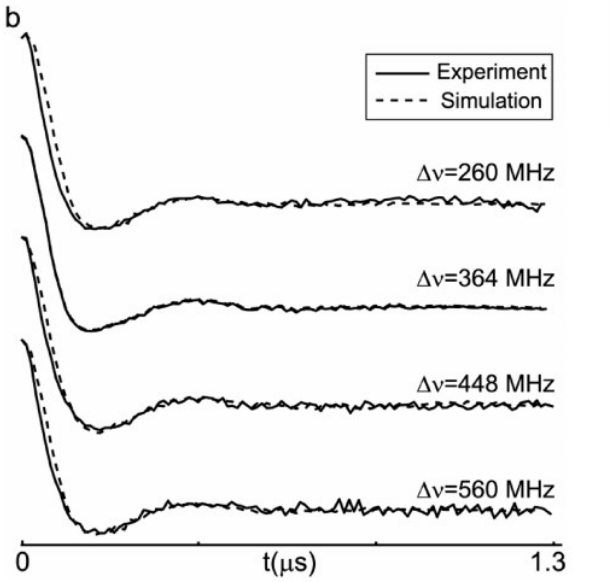


Plot and PC curve breakdown from fig4 set 1 "Conformational flexibility of nitroxide biradicals determined by X-band PELDOR experiments" By: D. Margraf, et al. 2007. See below for actual fig (left):





Plot and PC curve breakdown from fig3 ''An Approach towards the Measurement of Nanometer Range Distances Based on Cu2+ Ions and ESR'' By: Zhongyu Yang, Drew Kise, and Sunil Saxena, 2009.
See below for actual fig:



The figure sizes and titles/axes components had minimal changes done to them. The major change

was in removing the normalization conditions, as we already normalized our data in the process of scraping it. This code is still incomplete however, and will be refined at a later date.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter, AutoMinorLocator)
from matplotlib.figure import figaspect
import numpy.linalg as la


SMALL_SIZE = 24
MEDIUM_SIZE = 10
BIGGER_SIZE = 12

plt.rc('font', size=SMALL_SIZE)  # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)  # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)  # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)  # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)  # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)  # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

rc('font', **{'family': 'serif', 'serif': ['Arial']})

# parameters for the plot
n_trace = 4
pc_used = 2


data = np.loadtxt('fig3_an_approach_towards_the.txt')
t = data[:, 0]
deer = data[:, 1:]
tmax = max(t)


color = ['k', 'r', 'g', 'b', 'm', 'y']
startingSHC = np.eye(n_trace)
for i in range(n_trace):
    plt.plot(t, deer[:, i] + i, color[i])


# %% PRINCIPAL COMPONENTS ANALYSIS
u, s, vh = la.svd(deer[:, :])
m = u[:, :n_trace] * s


w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))


for i in range(n_trace):
    plt.plot(t[:], i * 1.1 + m[:, i], 'r')
    plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

plt.title("principal components")
ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.title('average value and \n principle components \n (bottom to top)')
plt.tight_layout()


s_app = s
smax = pc_used  #the number of principle component curves added to the curves
s_app[smax:n_trace] = 0
m_app = u[:, :n_trace] * s_app
deer_app = np.dot(m_app, vh)
shuffle = np.arange(n_trace)

w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))
for i in range(n_trace):
    plt.plot(t[:], shuffle[i] * 1.1 + deer_app[:, i], 'k')
    plt.plot(t[:], shuffle[i] * 1.1 + deer[:, i], 'r')

ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.title('fit with ' + np.str(smax) + '\n principle components')
plt.tight_layout()

# %% CENTERED PCA
xmean = np.mean(deer, axis=1)
X = deer.transpose() - xmean.transpose()
X = X.transpose()
u, s, vh = la.svd(X)
m = u[:, :n_trace] * s

w, h = figaspect(1.5)
```

```
fig = plt.figure(figsize=(w, h))

for i in range(n_trace):
    plt.plot(t[:], i * 1.1 + m[:, i], 'r')
    plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

plt.plot(t[:], -1.1 + xmean, 'r')
plt.plot(t[:], -1.1 + 0 * t[:], 'k', linewidth=0.5)
plt.title("principal components")
ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.tight_layout()
```

Modifications of PCA Code

22 March 2021 17:44

Aims:

- To modify the PCA code to accept the file inputs directly outputted from the cubic spline code.
- To modify the PCA code to save the m matrices for both centred and non-centred principle components, and also the deep_app variable incase it is required later.

Outcome:

The code has been modified such that it can read the output variable of the cubic spline script. The file names must be written at the top of the code in the same manner as they are defined at the top of the cubic spline code such that the script is able to see the text files. At the top of the code the output file names for the three outputs can also be modified.

The positioning of the graphs (whereby 1.1 is added to the amplitudes in each iteration) is not saved in these files and hence if this data is loaded and plotted these will need to be added so that the graphs are viewable properly.

The updated script can be found below.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter, AutoMinorLocator)
from matplotlib.figure import figaspect
import numpy.linalg as la

# File names and variable separation parameter
fileStr = 'deer_output.txt'
separationstr = '   '
saveStr_pc = 'prinicple_component_output.txt'
saveStr_centre = 'centred_principle_output.txt'
saveStr_deer_app = 'deep_app.txt'



SMALL_SIZE = 24
MEDIUM_SIZE = 10
BIGGER_SIZE = 12

plt.rc('font', size=SMALL_SIZE)  # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)  # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)  # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)  # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)  # fontsize of the tick labels
```

```
plt.rc('legend', fontsize=SMALL_SIZE)  # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

rc('font', **{'family': 'serif', 'serif': ['Arial']})

# parameters for the plot
n_trace = 2
pc_used = 2



#fileStr = 'deer_output.txt'
#fileRead = open(fileStr, 'r').readlines()
data = np.loadtxt(fileStr, delimiter=separationstr, skiprows=1)

t = data[:, 0]
deer = data[:, 1:]
tmax = max(t)



color = ['k', 'r', 'g', 'b', 'm', 'y']
startingSHC = np.eye(n_trace)
for i in range(n_trace):
   plt.plot(t, deer[:, i] + i, color[i])



# %% PRINCIPAL COMPONENTS ANALYSIS
u, s, vh = la.svd(deer[:, :])
m = u[:, :n_trace] * s



w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))



for i in range(n_trace):
   plt.plot(t[:], i * 1.1 + m[:, i], 'r')
   plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
```

```
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.title('average value and \n principle components \n (bottom to top)')
plt.tight_layout()



s_app = s
smax = pc_used  #the number of principle component curves added to the curves
s_app[smax:n_trace] = 0
m_app = u[:, :n_trace] * s_app
deer_app = np.dot(m_app, vh)
shuffle = np.arange(n_trace)

w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))
for i in range(n_trace):
   plt.plot(t[:], shuffle[i] * 1.1 + deer_app[:, i], 'k')
   plt.plot(t[:], shuffle[i] * 1.1 + deer[:, i], 'r')

ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.title('fit with ' + np.str(smax) + '\n principle components')
plt.tight_layout()

# Saves the m matrix to a file along with the time values
pltmatrix = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
   for n in range(n_trace+1):
      if n == 0:
         pltmatrix[i, n] = t[i]
      else:
         pltmatrix[i, n] = m[i, n-1]
np.savetxt(saveStr_pc, pltmatrix, delimiter=separationstr)
```

```
# Saves the deer matrix to a file along with the time values
pltmatrix_deer = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
   for n in range(n_trace+1):
      if n == 0:
         pltmatrix_deer[i, n] = t[i]
      else:
         pltmatrix_deer[i, n] = deer_app[i, n-1]
np.savetxt(saveStr_deer_app, pltmatrix_deer, delimiter=separationstr)

# %% CENTERED PCA
xmean = np.mean(deer, axis=1)
X = deer.transpose() - xmean.transpose()
X = X.transpose()
u, s, vh = la.svd(X)
m_c = u[:, :n_trace] * s

w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))

for i in range(n_trace):
   plt.plot(t[:], i * 1.1 + m_c[:, i], 'r')
   plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

plt.plot(t[:], -1.1 + xmean, 'r')
plt.plot(t[:], -1.1 + 0 * t[:], 'k', linewidth=0.5)
plt.title("principal components")
ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.tight_layout()

# Saves the centred m matrix to a file alogn with the time values
pltmatrix_c = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
```

```
    for n in range(n_trace+1):
        if n == 0:
            pltmatrix_c[i, n] = t[i]
        else:
            pltmatrix_c[i, n] = m_c[i, n-1]
np.savetxt(saveStr_centre, pltmatrix_c, delimiter=separationstr)
```

Data scraping

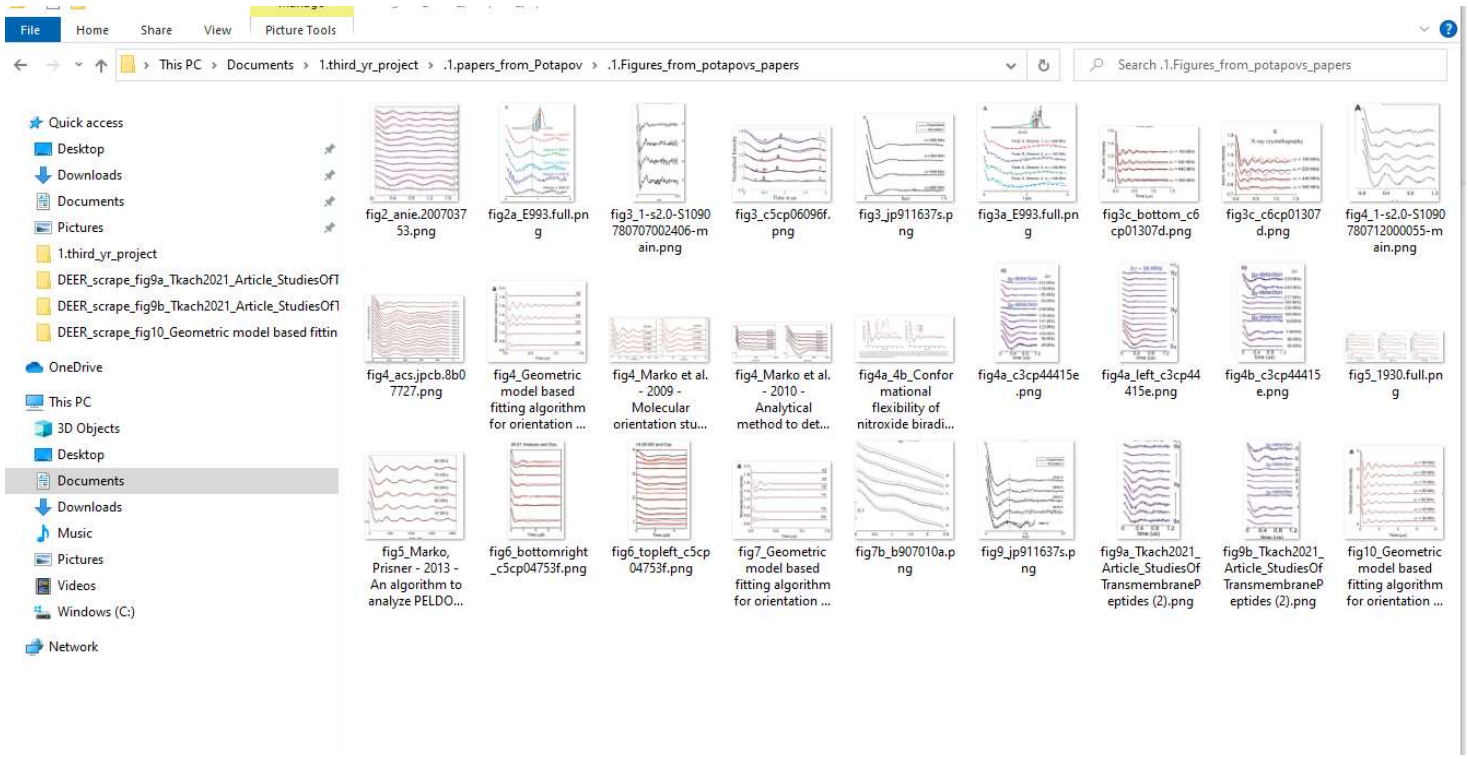
24 March 2021 11:54

Aims:

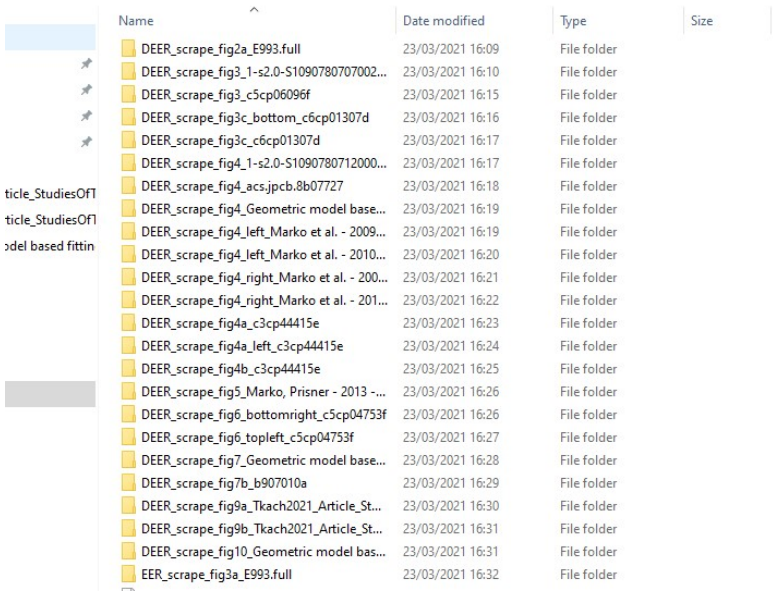
- To scrape the data from the figures contained in the list of papers Dr. Potapov supplied.
- Run these data sets through the cubic spline script.

Outcomes:

Of the 19 papers supplied only 17 contained relevant/usable figures, see below:



Data was scraped from all of these figures, and placed into txt files and organised accordingly:



After the txt files were organised, each data set from each corresponding figure was ran through the cubic spline script as appears in previous entry, and the cubic spline for each figure was obtained. These cubic spline files will be added to the table containing the papers.

The next steps will be to run these cubic spline scripts through the SVD script to obtain the principle curves.

MPP Weight Code

06 April 2021 17:41

Aims:

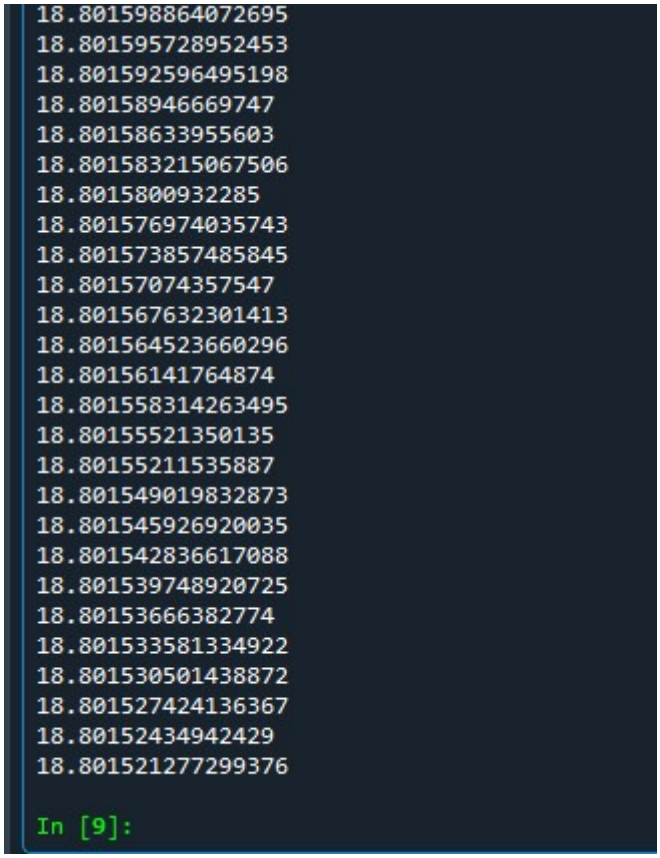
- To look at the part of the code that deals with the MPP weights and distance distribution
- To modify this code to work with n many inputs and n many principle components, with the final goal being to allow the code to find the distances
- Verify that the code works using the root-mean-square difference between the theoretical and experimental datasets by seeing if they converge as expected, and by seeing if the distance distribution appears correctly

Outcome:

The code has been modified such that it reads in the data files, and can be provided with n many traces and n many principle components that were found from these traces with the previous script. Some variables, such as the colours of the plotted traces, had to be omitted to allow for the n many system to work.

The code does not currently produce distance distributions properly – I do believe this is something to do with a misinterpretation of how the custom function gaussian function works, written as part of Dr. Potapov's orisel library. I intend to ask about the workings of this function in the next project meeting.

The root-mean-square difference between the theoretical and experimental datasets do appear to converge correctly in console, as they are very similar.



Running the code with the linspace of r between 2 (number of principle components for the dataset for which this code was written) and 6 (number of traces for the dataset for which this code was written) shows a distance distribution that is curved, however I am not sure if this is correct as it does not look right as it should have a peak – I will inquire at the next project

meeting.
Using this linspace from the n many components to the n many traces gives a straight line distance distribution, which is incorrect.

Cubic Splined Data Analysed (PCA)

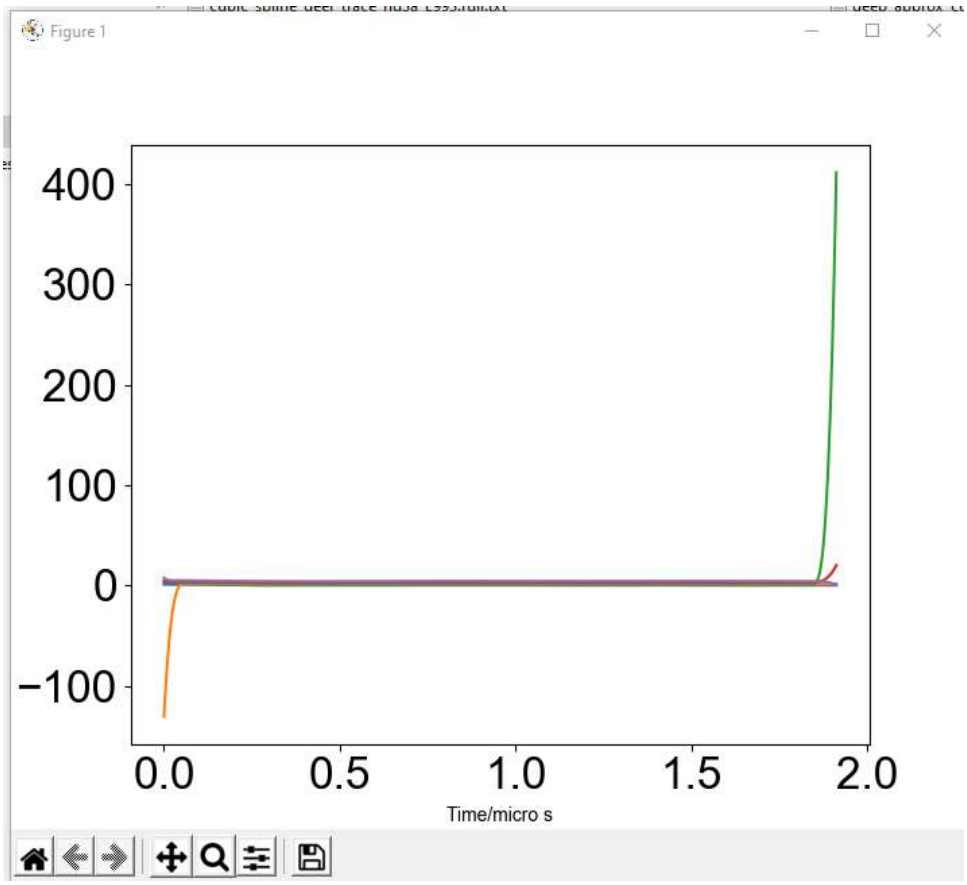
10 April 2021 18:17

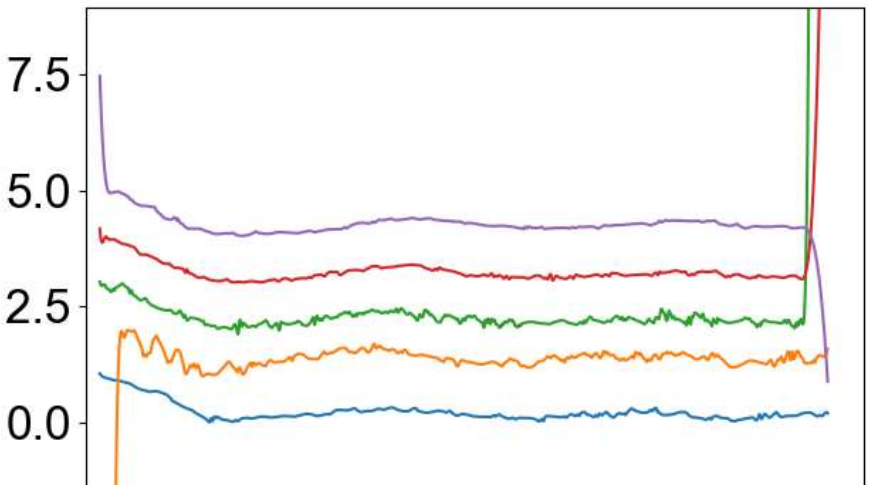
Aims:

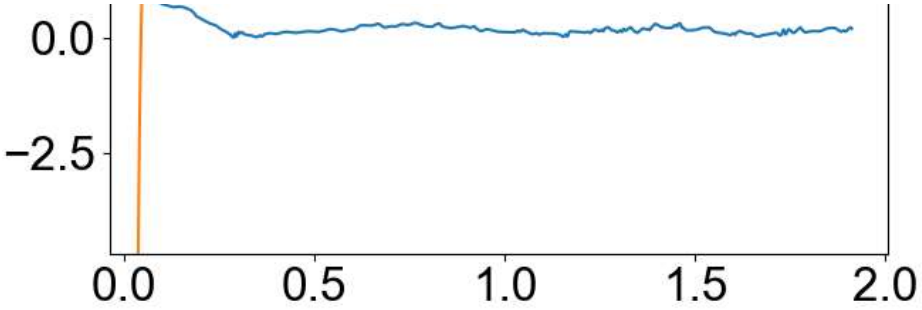
- To take the cubic splined data from the data scraped from the table containing paper references, and run them through the PCA script to find the principle curves for the data sets.
- From the produced PC curves deduce how many PC curves are required to accurately reproduce their corresponding data sets.

Outcomes:

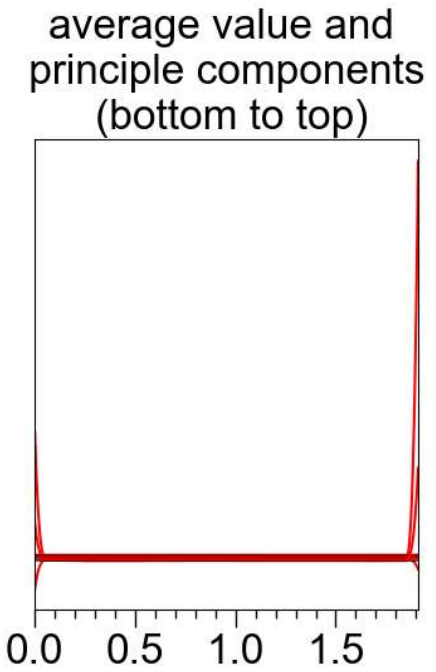
In the table containing all of the papers and figures, there is a column containing the numbers of PC curves used to build up the data. For some of these scraped data sets there were some issues with the cubic spline that made some points go off to infinity and hence skew the mean PC data, see below:

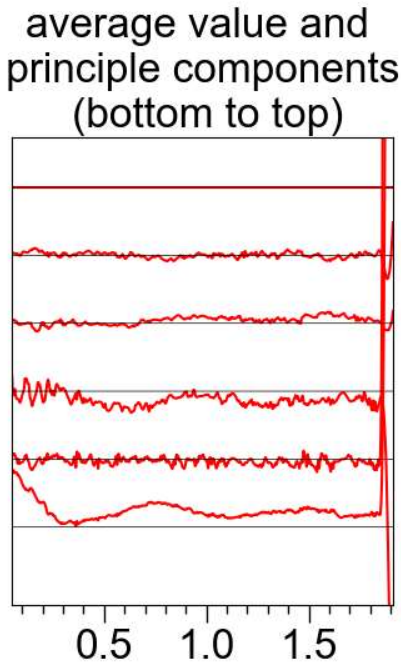






As can be seen there must have been some portions of the data that finished to early/late with respect to the others, this also had a negative effect on the useful output data for the PCA:





For the majority of the scraped data this was not an issue though there are around 4 data sets that will need re-scraping. To see the PCA curves that worked, see the table of papers.

MPP Code – Point Quantity and Testing

11 April 2021 20:48

Aim:

- To change the MPP Weight code to allow the algorithm which reads data from the file to skip over data points, making edits in the code faster to test. The data has already been scraped with a high number of points, so editing the code in this way makes more sense time-wise than re-scraping the data.
- To test the MPP Weight code with the dataset from Dr. Potapov's paper to see if a correct distance distribution can be seen.

Outcome:

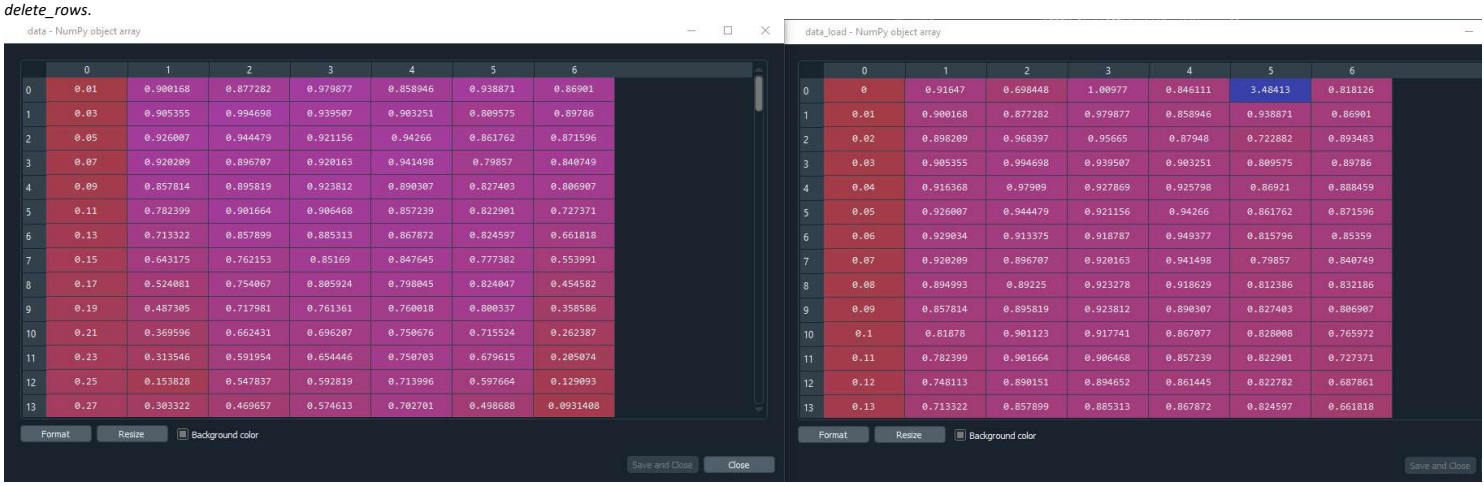
The data input correctly skips over every x rows, with x being defined by the user. The algorithm used to do this is:

```
def delete_rows(mat, spacing):
    n_r = len(mat)/spacing
    n_c = len(mat[0])
    new_mat = np.zeros((int(len(mat)-n_r),int(n_c)))

    j = 0
    for i in range(len(mat)):
        if i % spacing != 0 or i == 0:
            for n in range(n_c):
                print(str(i) + ", " + str(j))
                new_mat[j-1, n] = mat[i, n]
            j += 1

    return new_mat
```

The effects of this can be seen on these 2 matrices – with the matrix *data_load* being the matrix loaded from the text file, and the matrix *data* being the output from the above algorithm *delete_rows*.



data - NumPy object array

	0	1	2	3	4	5	6
0	0.01	0.900168	0.877282	0.979877	0.858946	0.938871	0.86901
1	0.03	0.905355	0.994698	0.939507	0.903251	0.809575	0.89786
2	0.05	0.926007	0.944479	0.921156	0.94266	0.861762	0.871596
3	0.07	0.920209	0.896707	0.920163	0.941498	0.79857	0.840749
4	0.09	0.857814	0.895819	0.923812	0.890307	0.827403	0.806907
5	0.11	0.782399	0.901664	0.906468	0.857239	0.822901	0.727371
6	0.13	0.713322	0.857899	0.885313	0.867872	0.824597	0.661818
7	0.15	0.643175	0.762153	0.85169	0.847645	0.777382	0.553991
8	0.17	0.524081	0.754067	0.805924	0.798045	0.824047	0.454582
9	0.19	0.487305	0.717981	0.761361	0.760018	0.800337	0.358586
10	0.21	0.369596	0.662431	0.696207	0.750676	0.715524	0.262387
11	0.23	0.313546	0.591954	0.654446	0.750703	0.679615	0.205074
12	0.25	0.153828	0.547837	0.592819	0.713996	0.597664	0.129093
13	0.27	0.303322	0.469657	0.574613	0.702701	0.498688	0.0931408

Format Resize Background color Save and Close Close

data_load - NumPy object array

	0	1	2	3	4	5	6
0	0	0.91647	0.698448	1.00977	0.846111	3.48413	0.818126
1	0.01	0.900168	0.877282	0.979877	0.858946	0.938871	0.86901
2	0.02	0.898209	0.968397	0.95665	0.87948	0.722882	0.893483
3	0.03	0.905355	0.994698	0.939507	0.903251	0.809575	0.89786
4	0.04	0.916368	0.97909	0.927869	0.925798	0.86921	0.888459
5	0.05	0.926007	0.944479	0.921156	0.94266	0.861762	0.871596
6	0.06	0.929034	0.913375	0.918787	0.949377	0.815796	0.85359
7	0.07	0.920209	0.896707	0.920163	0.941498	0.79857	0.840749
8	0.08	0.894993	0.89225	0.923278	0.918629	0.812386	0.832186
9	0.09	0.857814	0.895819	0.923812	0.890307	0.827403	0.806907
10	0.1	0.81878	0.901123	0.917741	0.867077	0.828008	0.765972
11	0.11	0.782399	0.901664	0.906468	0.857239	0.822901	0.727371
12	0.12	0.748113	0.890151	0.894652	0.861445	0.822782	0.687861
13	0.13	0.713322	0.857899	0.885313	0.867872	0.824597	0.661818

Format Resize Background color Save and Close

This image is the result of a 100-point dataset being loaded in and every second row ignored – it can be seen that the algorithm works correctly.

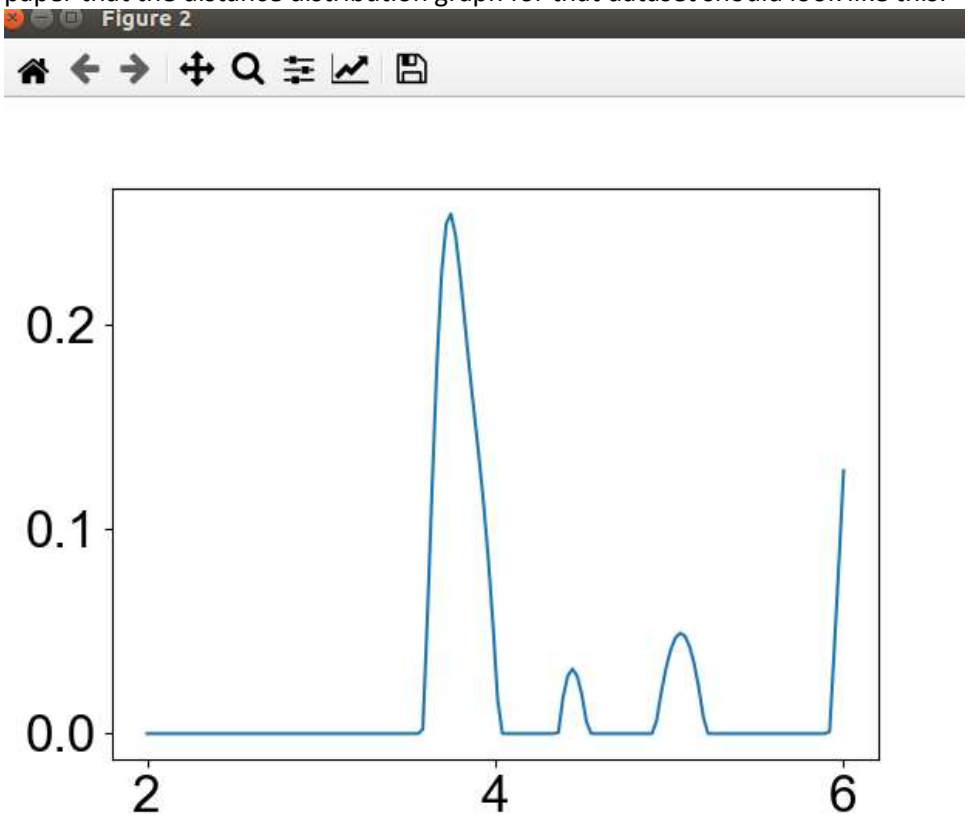
Next this 100-point halved dataset, from Dr. Potapov's paper, was used to test the MPP Weight code. It once again did not produce a correct looking distance distribution. I will consult Dr. Potapov at the next project meeting and ask for assistance understanding his original algorithm.

MPP Code – Distance Distribution

17 April 2021 21:34

Aim:

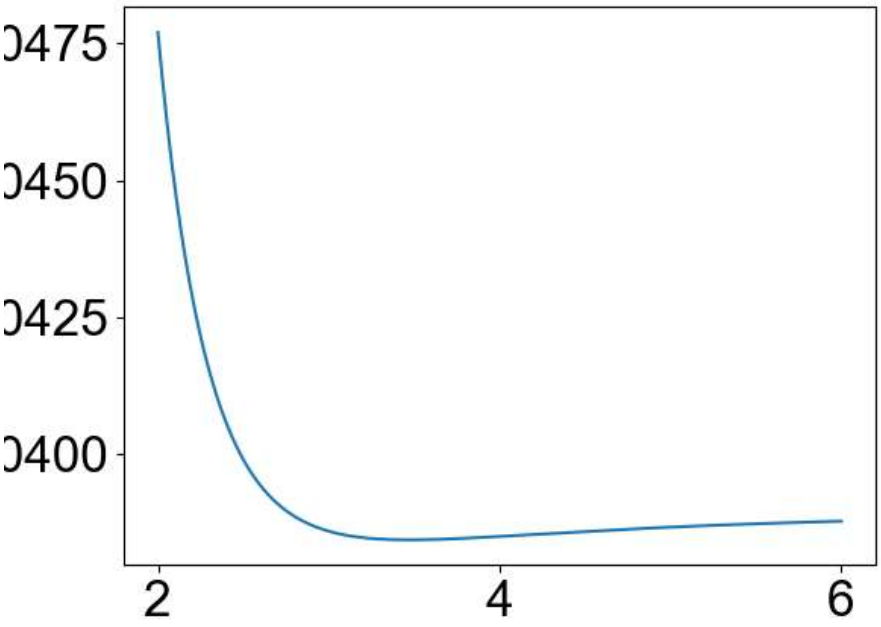
- To fix the distance distribution graph produced by the MPP weight code. In previous attempts, the graph had appeared as an exponential decay with a slight increase towards the end of the 6nm range – this is incorrect. It can be seen from the graph provided in Dr. Potapov's 2020 paper that the distance distribution graph for that dataset should look like this:



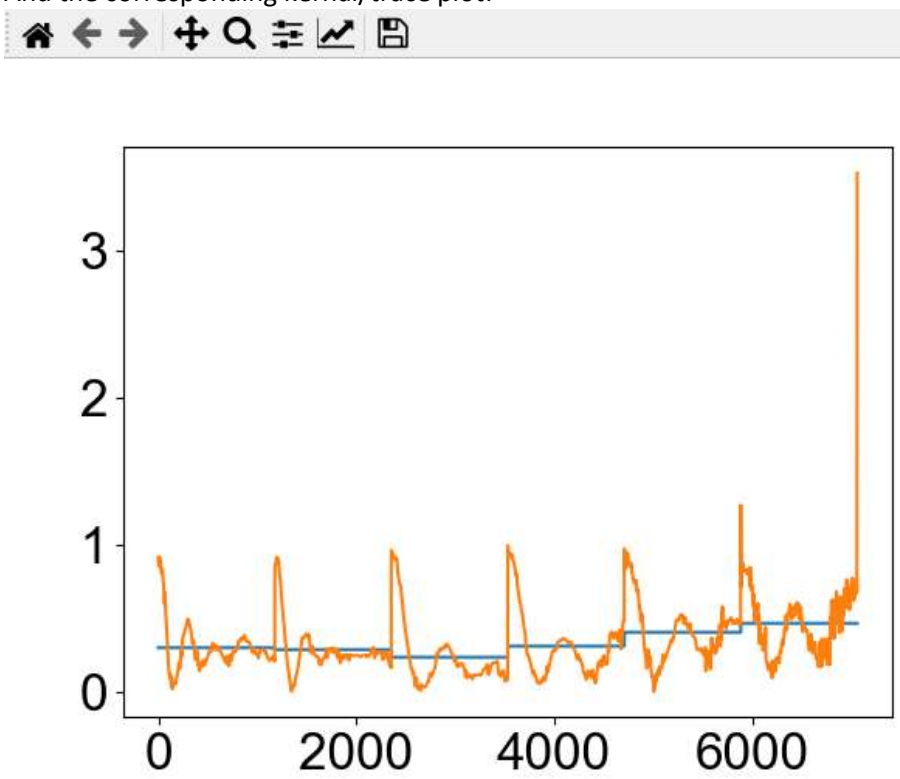
Our aim was to produce a similar looking graph by scraping the data from Dr. Potapov's paper. This could then be used to test the code, ensuring the distance distribution algorithm works correctly before applying it to the other scraped datasets.

Outcome:

The first time the code was run with Dr. Potapov's dataset, the code produced a distance distribution with the exponential decay shown below.



And the corresponding kernal/trace plot:

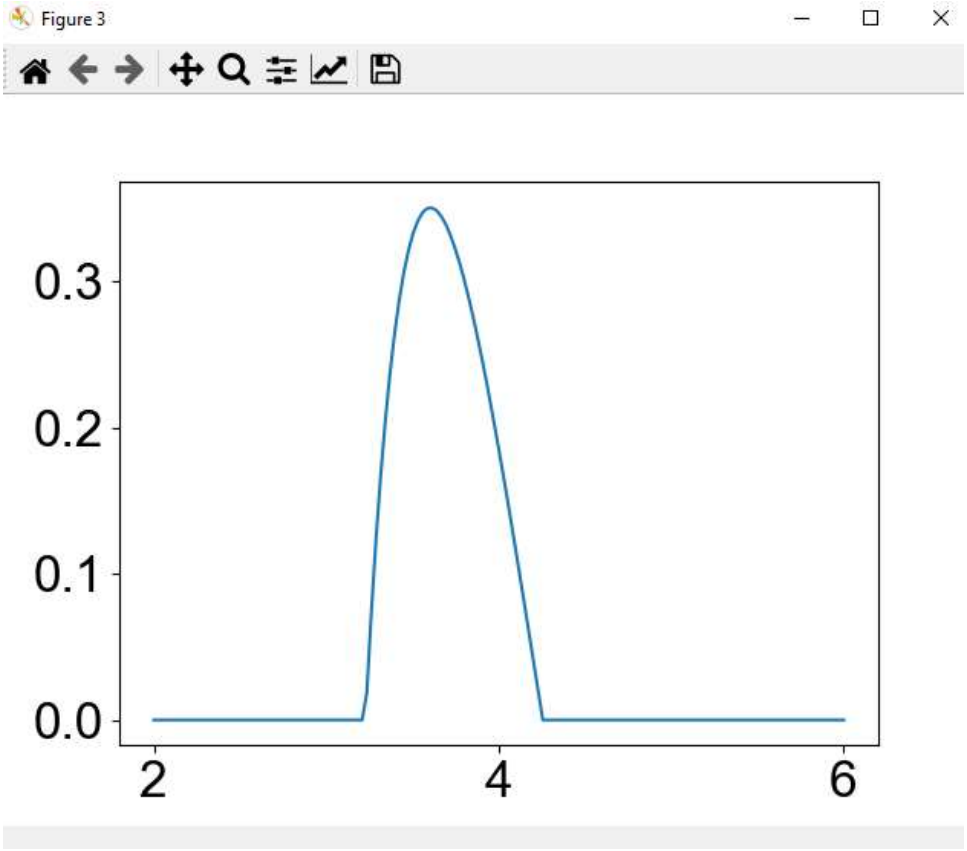


This distance distribution is clearly incorrect, and shows no peaks at all that can be used to calculate distance.

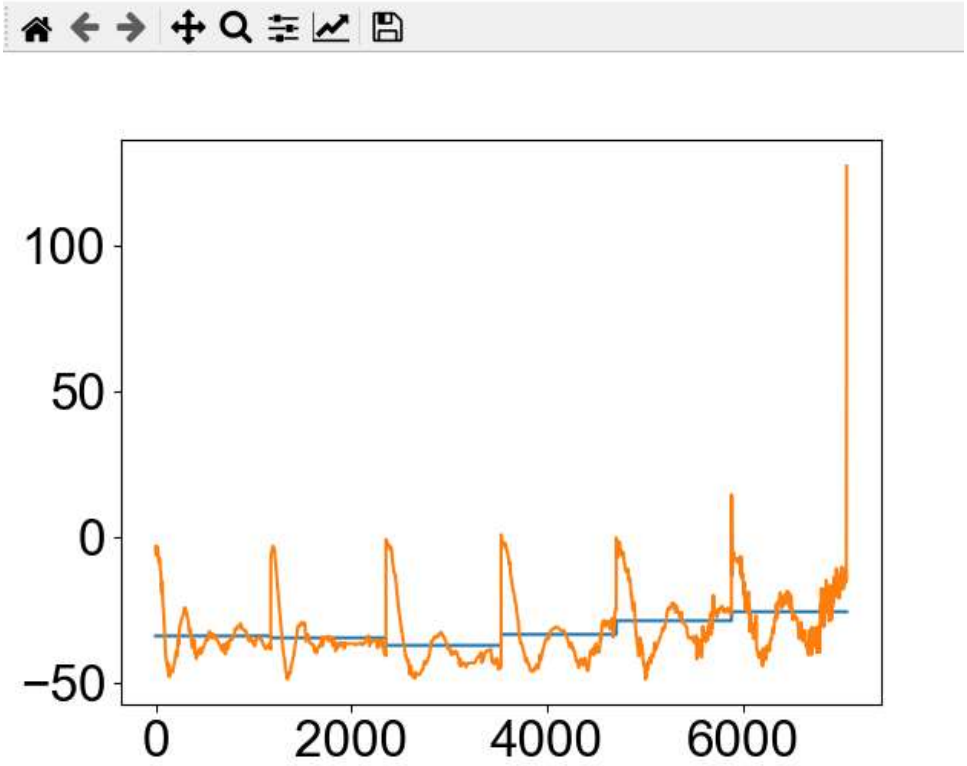
Our first potential thought was the normalisation factor – Dr. Potapov's code contained a normalisation factor, lambda, for each of the 6 traces. Because we require this code to run for n many traces, this part of the code was omitted. For testing's sake, the line of code which normalises the DEER traces was added back, taking an approximate rough average lambda value to see what effect this would have on the code (with the rough approximate lambda being 0.02).

```
fit = np.exp(-t*1e-3*0.02)
for i in range(n_trace):
    deer[:,i] =  (i_deer[:,i]/fit - 0.98)/0.02
```

This had the following effect on the distance distribution and kernal/trace plots:



.

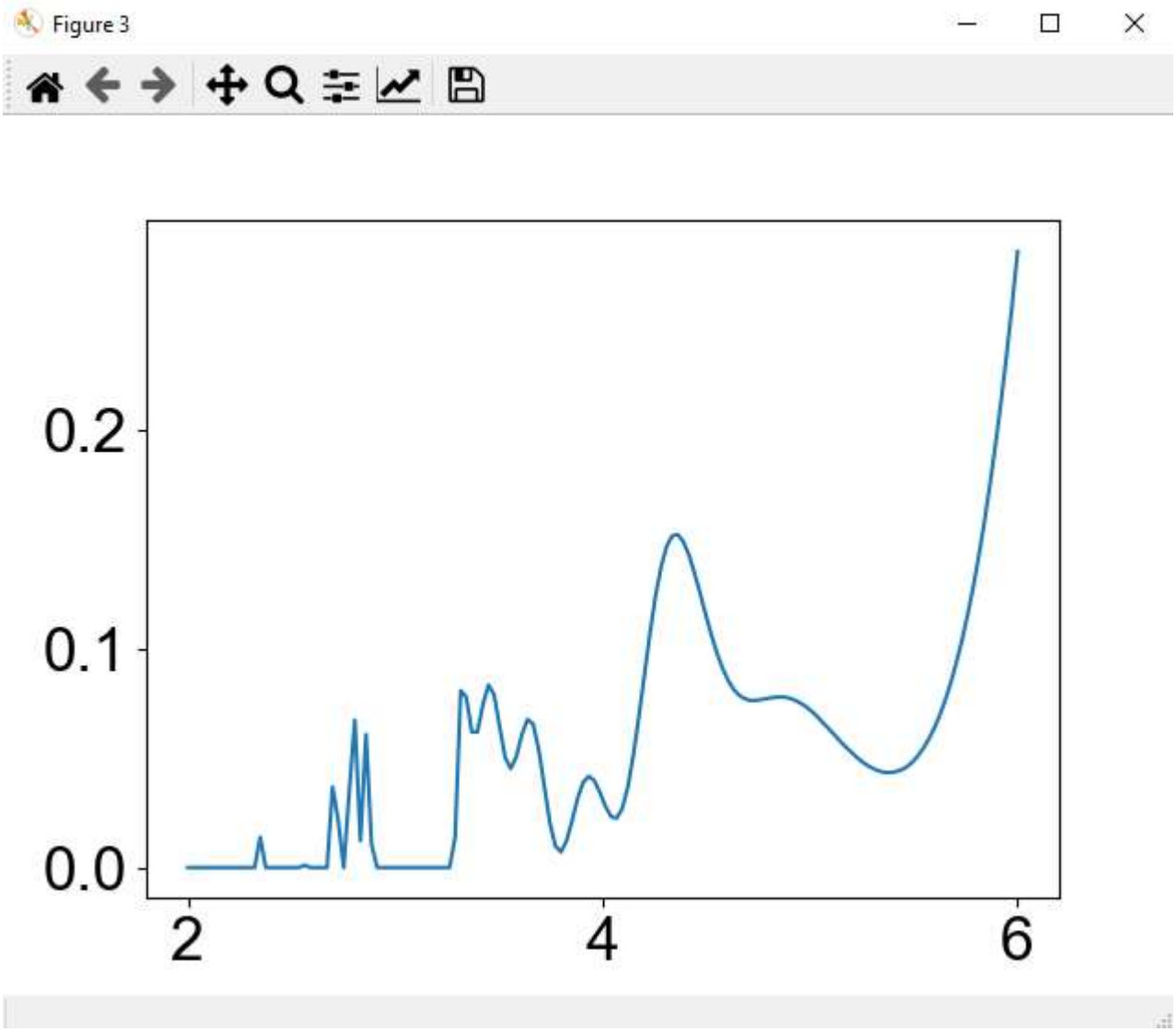


The distance distribution plot now featured a peak, which made it look more like a distance distribution and gives an actual measurable distance of around 3.5nm, which looks similar to the maximum peak of Dr. Potapov's distance distribution. It does not, however, feature the other peaks, and so is still an erroneous data distribution. In addition, the kernal/trace plot has been offset by a large factor of around –40 due to the normalisation.

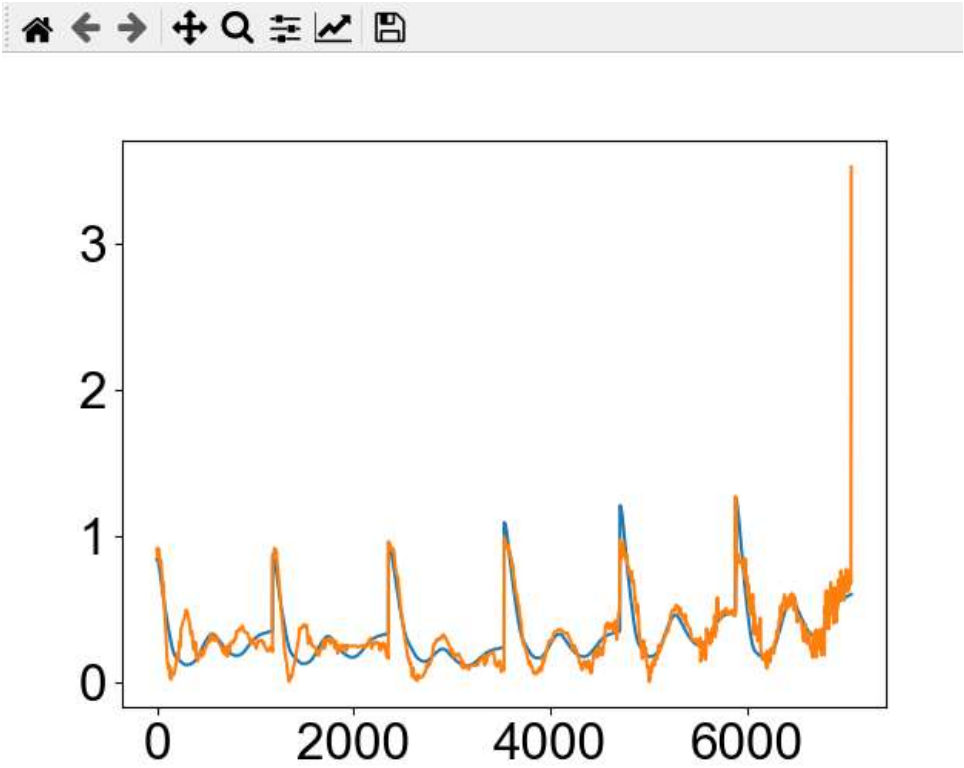
It is logical that this method did not produce useful results as the scraped data was already output

from Dr. Potapov's code as normalised data. It can also be seen that the line with which the traces tend to on the kernal/trace plot is not zero as it should be – this is due to the data recorded being taken from a maximum of 1 to a minimum of 0. The next two potential fixes we attempted were to try re-scraping this data such that it tended to and oscillated around 0 at the end, and to further examine the normalisation factors in the code.

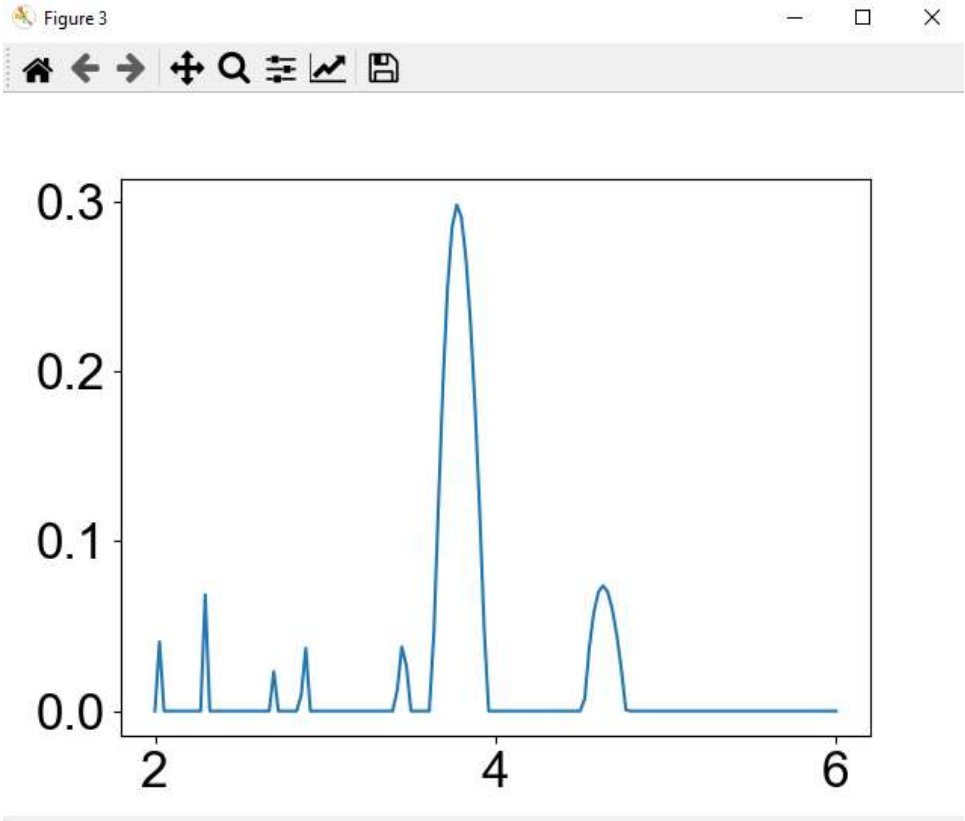
After looking through potential normalisation factor, a factor of 1e-3 was discovered in Dr. Potapov's orisel.py library in the make_kernal function. Omitting this caused the code to produce a much better looking distance distribution, and a much closer matching kernal/trace plot:



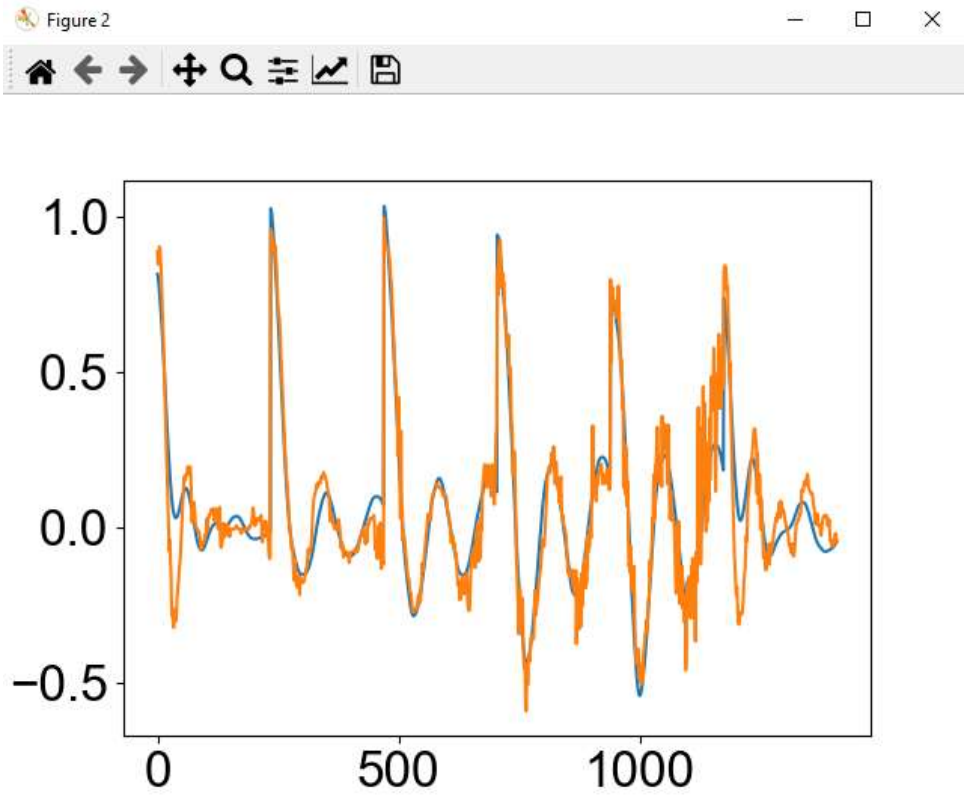
.



This is likely due to normalisation discrepancy between our scraping Dr. Potapov's already normalised traces as mentioned above.
The issue with this new distance distribution was the fact that it seems to steadily increase in the y-axis, and the highest peak is no longer at roughly 3.5nm like Dr. Potapov's highest peak.
The next test to attempt was a rescrape of Dr. Potapov's data such that the traces tend to oscillate around 0 at their end, and then to pass this newly scraped data through the code with the fixed normalisation. This was the result:



.



This produces the best distance distribution by far, and the kernal/trace plot fits well. It does differ from Dr. Potapov's, but this is likely due to the inaccuracies of scraping the data. The peak appears to be in around the correct place in comparison to Dr. Potapov's distance distribution.

The next task to do is to acquire a distance distribution for all of the other datasets. It will be necessary to ensure these datasets' traces also tend to and oscillate around 0 at their end. We thought of three potential ways of doing this: coding an algorithm to move the traces down, manually moving each of the traces down in the code by eye, or rescraping all of the datasets in the correct manner. We have decided that the least time consuming method is to rescrape all the data.

It is worth noting to achieve the best distance distribution plot we had to modify Dr. Potapov's orisel.py library slightly. We made the following modification to the *make_kernal* function:

```
def make_kernel(t, r, SHC):
    Kernel = np.zeros((len(t),len(r)))
    K0 = DEER(SHCs = SHC)
    for i in range(len(r)):
        r1 = np.array([r[i]])
        f1 = np.array([1])
        Kernel[:,i] = K0.trace(r1, f1, t/1)
    return Kernel
```

Our modification can be seen on the second to last line, where instead of it originally reading *Kernal[:,I] = K0.trace(r1, f1, t/1e3)* it now reads *Kernal[:,I] = K0.trace(r1, f1, t/1).* The division by 1 has been left in the function to show where it has been modified.

MPP Distance Calculation

20 April 2021 16:41

Aim:

- To produce an algorithm for calculating the distance at the peak of the distance Distibution produced by the MPP code so that data can be collected.

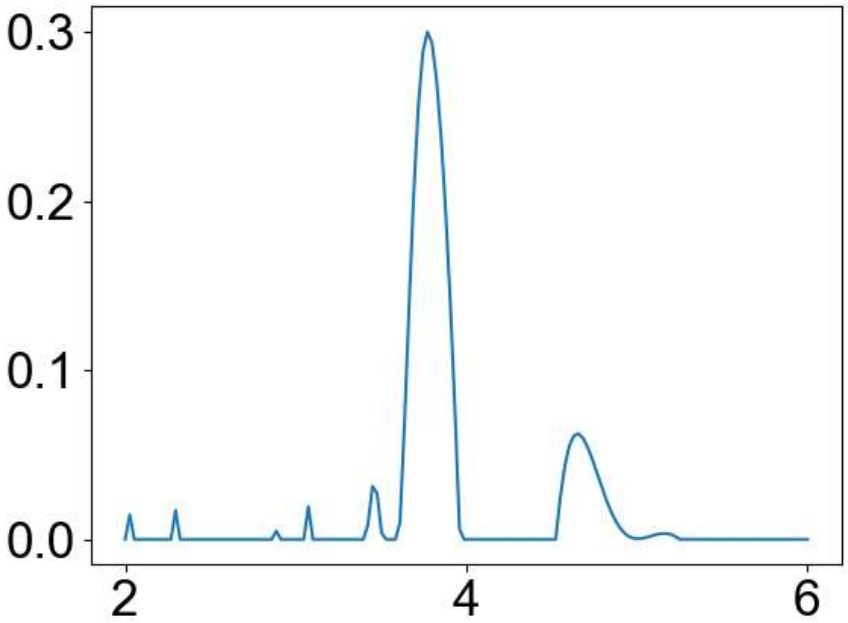
Outcome:

The piece of code that was written for the calculation of the distance is:

```
x_max_index = 0
x_max = max(x)
for i in range(len(x)):
    if x[i] == x_max:
        x_max_index = i
print('Distance distribution peak at ' + str(r[x_max_index]) + 'nm.')
```

This algorithm finds the maximum x (which is f(r)) value, finds the index of this value in the x array, and then finds the r value at the corresponding index in the r array, hence finding the distance. It then prints this.

The code was run using Dr. Potapov's data, producing this distance distribution plot:



The algorithm printed the following line using this data:

```
Distance distribution peak at 3.771812080536913nm.
```

This appears to be consistent with the distance distribution plot as the peak does appear to be around the 3.8 position, hence the calculation code appears to work as intended.

The MPP code is now finished, and can be used for the final data collection.

Distance Distribution Data

21 April 2021 20:12

Aim:

- To gather the distance data from the distance distributions produced using the MPP weight code.

Outcome:

The data collected was placed into the following table:

Paper	Data '.txt' files	Distance from Paper (nm)	Our Distance	Comments	Number of Traces	Number of PC Curves	Field/Protein Structure
Potapov, fig 3c	 DEER_center0	3.75 +- 0.13	3.765100671140939 6nm	Figure 3c	6	3	3374.7 to 3385.2 mT/ Nitroxide biradical molecule
 anie.200703753	 DEER_scrape_fig2_an...	3.25	3.281879194630872 3nm	Figure 2a	13	9	6.4045 to 6.4225T/Tyrosyl biradical attached to a RNR R2 dimer.
 E993.full	 DEER_scrape_fig2a_E...	3.5 +- 0.1	2.449 nm	This data set has too much noise, unable to discern the actual value from the noise. Figure 2a Data was cut to avoid error caused by spline	5	4	3090 to 3342 x10^-4 T / Cu2+ EcoRI-DNA Complex

				method .			
1-s2.0-S109 07807070...	DEER_scrap e_fig3_1-...	2.2 +- 0.3	2.8791946 30872483 nm	Note – another high peak at ~ 2.04nm , possibl e noise Figure 3 Data was cut to avoid error caused by spline method .	4	4	3195 to 3315 x10^-4 T / Cu2 Poly-prolene peptide
c5cp06096f	DEER_scrap e_fig3_c5...	3.62±0.05	2.7449664 4295302n m	3.4698 nm is another high peak much closer to exact value could be caused by noise Figure 3 Data was cut to avoid error caused by spline method .	5	3	X-BAND / Cu2 ARTHROBACTA GLOBIFORMIS
c6cp01307 d	DEER_scrap e_fig3c_c...						

c6cp01307d	DEER_scrape_fig3c_c... DEER_scrape_fig3c_b...	2.25 (viewed from graph, doesn't stat, fig3 a (T21R1) (Top) 2.1 (viewed from graph) (Bottom) (T30R1)	2.100671140939597 nm (Top) 2.087248322147651 nm (Bottom)	Figure 3c	4	2	X-Band Top: Cu2+NO (radical), T21R1 Bottom:Cu2+NO (radical) T30R1
1-s2.0-S10907807120...	DEER_scrape_fig4_1-...	2.65±0.1	2.637583892617449 5nm	Figure 4	5	2	X-BAND / VT Crystal structure ribbon, N-O radicals
acs.jpcb.8b07727	DEER_scrape_fig4_ac...	2.55 +- 0.5	2.395973154362416 nm	Figure 4	17	5	11020 to 11774 x10-4 / Cu2 (radical)
Geometric model ba...	DEER_scrape_fig4_ge...	1.93 +- 0.03	2.449664429530201 nm	Figure 4	6	3	W-BAND / Nitroxite radical,
Marko et al. - 2010 ...	DEER_scrape_fig4_lef... DEER_scrape_fig4_rig...	Paper not specifying distance N/A	2.476510067114094 nm (left) 3.375838926174496 5nm (right)	Figure 4	6	3	X-BAND / Nitroxide biradical, DNA(1,5)(LEFT), DNA(1,12)(Right)
Marko et al. - 2009 ...	DEER_scrape_fig4_lef... DEER_scrape_fig4_rig...	~3.3 (Read from graph) ~3.3 (read from graph)	3.308724832214765 nm (left) 3.295302013422819 nm (right)	Figure 4	5	2	X-BAND / Nitroxide biradical (linear) (left), Nitroxide biradical (bent) (right)

c3cp44415e	DEER_scrape_fig4a_c... DEER_scrape_fig4a_l...	3.1±0.3 (both)	3.255033557046979 7nm (left) 3.134228187919463 nm (right)	Figure 4a	11 (left) 14 (right)	9 (left) 10 (right)	W-BAND / Nitroxide radicals, RNA duplex Data file that says left is right graph
c3cp44415e	DEER_scrape_fig4b_c...	2.8±0.2	1.765100671140939 6nm	Figure 4b Data was cut to avoid error caused by spline method. Very clear noise from the 363MHz trace, unlikely to be good data.	10	4	W-BAND / Nitroxide radicals,Alpha helical peptide
Marko, Prisner - 2...	DEER_scrape_fig5_M...	2.7	2.69nm	Figure 5 Initial 20 points cut to avoid error caused by spline method. Quite noisy.	5	3	X-BAND / Nitroxide birdadicals
c5cp04753f	DEER_scrape_fig6_bo...						

c5cp04753f	DEER_scrape_fig6_bo... DEER_scrape_fig6_to...	~3.5 (Read from graph) (Bottom right) ~5 (Read from graph) (Top left)	3.657718120805369nm (Bottom right) 4.201342281879195nm (Top left)	Figure 6	7 (Bottom right) 13 (Top left)	3 (Bottom right) 6 (Top left)	W-BAND / Nitroxide, RX26-27(bottom right), RX19-20 (top left)
Geometric model ba...	DEER_scrape_fig7_Ge...	1.8+-0.02	2.630872483221476 3nm (2.61073825503355 7nmTEMP)	Figure 7	6	4	W-BAND / Nitroxide biradical
Tkach2021_Article_S...	DEER_scrape_fig9a_T...	2.8±0.2	3.456375838926174 3nm	Figure 9a (Fixed frequency)	10	7	56 MHz / Alpha rich, alpha top (biradical), peptide
Tkach2021_Article_S...	DEER_scrape_fig9b_T...	2.8±0.2	3.456375838926174 3nm	Figure 9b (Varied frequency)	10	7	W-BAND / Alpha rich, alpha top (biradical), peptide
Geometric model ba...	DEER_scrape_fig10_...	3.46±0.6	3.738255033557046 6nm	Figure 10 Another peak at 3.2953 nm. Distance distribution is noisy.	7	5	X-BAND / Trityl nitroxide biradical
E993.full	EER_scrape_fig3a_E9...	2.2±0.2 and 4.2± 0.3 (paper states two distances)	2.167785234899328 7nm	Figure 3a Data was cut 65 points from the end to avoid	4	3	X-BAND / Cu^2+-nitroxide (radical), onMTSSL-Ser180Cys-EcoRI-DNA complex

				error caused by spline method. There are also peaks at ~ 2.48nm (within a large amount of noise) and ~ 3.98nm (clearer peak), the second of which could also match the data. Distance distribution quite noisy.			

Filling in the data table (from previous entry)

28 April 2021 22:53

Aims:

- To gather the field strength/ frequency data for the scraped DEER traces and add them to the data table.
- To gather the distance data for the scraped DEER traces and add them to the data table, and write the paramagnetic species used for the DEER.

Outcomes:

The DEER trace field data was gathered and added to an additional column of the data table as well as the field strength/frequencies used to produce the DEER traces and finally the type of paramagnetic species used.

For most cases this was a straight forward process, though in some the distance values were not stated and were shown only in graphical form and so there will be some error in our interpretation of these distances. Also in some cases it was not clear what type of paramagnetic species was used for what specific DEER trace figure, though this was very rarely an issue.

Cubic Spline Code - Final

09 May 2021 16:07

This is the code used for the cubic spline part of the project in its final form.

```
from scipy.interpolate import CubicSpline
import numpy as np
import collections
import os.path as op
import matplotlib.pyplot as plt
from numpy.linalg import inv

# Number of graphs
g_count = 6
# Optional Curve labels - leave blank for automatic labelling
labels = ['Curve 1', 'Curve 2']



# String to separate numerical values
separationstr_wr = "   "
separationstr_r = "   "

# Creating the column labels for the output file
if len(labels) != g_count:
   autolabels = []
   for i in range(0, g_count):
      autolabels.append("A_" + str(i+1))
else:
   autolabels = labels

# Puts the data into a matrix, ensuring the matrix can be created by filling blank spaces with NaN values
def initialise_data(t, y, n):
   global t_m
   global a_m

   for i in range(0, len(t)):
      t_m[n, i] = t[i]
      a_m[n, i] = y[i]
```

```
    if len(t) < pt_max:
        for i in range(len(t), pt_max):
            t_m[n, i] = np.NaN
            a_m[n, i] = np.NaN

def file_len(fname):
    with open(fname) as f:
        for i, l in enumerate(f):
            pass
    return i + 1

# Builds the row strings for printing into the file
def build_wr_str(t, a, n):
    global g_count
    global separationstr_wr

    wr_str = str(t[n]) + separationstr_wr

    for i in range(0, g_count):
        wr_str = wr_str + str(a[i])
        if i != g_count-1:
            wr_str = wr_str + separationstr_wr
        else:
            wr_str = wr_str + "\n"

    return wr_str

# Splits a string at a specific index
def split_string(str_i, sep, n):
    sep_ind = str_i.find(sep)
    if n == 1:
        return str_i[0:sep_ind]
    else:
        return str_i[(sep_ind + len(sep)):(len(str_i) - 1)]

tmin = 0
tmax = 0
pt_max = 0

# Find the maximum value of points
for i in range(0, g_count):
```

```
    fileStr = '1.DEER_center0_' + str(i+1) + '.txt'
    count = file_len(fileStr)

    if count > pt_max:
        pt_max = count

# Set up the data matrices
t_m = np.zeros((g_count, pt_max))
a_m = np.zeros((g_count, pt_max))

# Reads the data into arrays, and by calling the initialise function adds the data to the matrices
for i in range(0, g_count):
    readData_t = []
    readData_a = []
    fileStr = '1.DEER_center0_' + str(i+1) + '.txt'
    fileRead = open(fileStr, 'r').readlines()
    for n in range(0, len(fileRead)):
        readData_t.append(float(split_string(fileRead[n], separationstr_r, 1)))
        readData_a.append(float(split_string(fileRead[n], separationstr_r, 2)))

    if max(readData_t) > tmax:
        tmax = max(readData_t)

    initialise_data(readData_t, readData_a, i)


# Plots the data
ts = np.arange(tmin, tmax, 0.001)
plt.figure()
for i in range(g_count):
    plt.plot(t_m[i,:], a_m[i,:] + i)

splines = []

# Performs the cubic spline for all of the data
for n in range(0, g_count):
    current_set_t = t_m[n,:].tolist()
    current_set_a = a_m[n,:].tolist()

    for i in range(pt_max-1, 0, -1):
```

```
        if np.isnan(current_set_t[i]):
            del current_set_t[i]
            del current_set_a[i]

    current_set_t.sort()

    #Finds the index at which a duplicate is initially found
    repeatx = [item for item, count in collections.Counter(current_set_t).items() if count > 1]
    while len(repeatx) > 0:
        ind = current_set_t.index(repeatx[0])

        #Loop repeats until there are no more duplicates
        current_set_a.remove(current_set_a[ind])
        current_set_t.remove(current_set_t[ind])
        repeatx = [item for item, count in collections.Counter(current_set_t).items() if count > 1]

    #Performing cubic spline
    splines.append(CubicSpline(current_set_t, current_set_a))



# Filewriting Part

fy = open("deer_output.txt", "a")
sp_af = np.zeros((g_count, int(tmax*1000)+1))

namestr = "TIME:" + separationstr_wr

for n in range(0, g_count):
    current_af = splines[n](ts)
    for i in range(0, len(ts)):
        sp_af[n,i] = current_af[i]

    if n == g_count -1:
        namestr = namestr + autolabels[n] + "\n"
    else:
        namestr = namestr + autolabels[n] + separationstr_wr

fy.write(namestr)

for i in range(0, len(ts)):
    print_vector = []
```

```
    for n in range(0, g_count):
        print_vector.append(sp_af[n,i])

    fy.write(build_wr_str(ts, print_vector, i))
fy.close()

plt.figure()
for i in range(g_count):
    plt.plot(ts, sp_af[i,:] + i)
```

PCA Code - Final

09 May 2021 16:08

This is the script for the PCA part of the project in its final form.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter, AutoMinorLocator)
from matplotlib.figure import figaspect
import numpy.linalg as la



### SETUP INFORMATION



# File names and variable separation parameter
fileStr = 'INSERT FILE NAME HERE.txt'
separationstr = '   '
saveStr_pc = 'prinicple_component_output.txt'
saveStr_centre = 'centred_principle_output.txt'
saveStr_deer_app = 'deep_app.txt'

# Font size parameters
SMALL_SIZE = 10
MEDIUM_SIZE = 16
BIGGER_SIZE = 24

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=BIGGER_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

rc('font', **{'family': 'serif', 'serif': ['Arial']})

# Parameters for the plot
```

```
n_trace = 6
pc_used = 3


### END OF SETUP INFORMATION


# Variable setup
data = np.loadtxt(fileStr, delimiter=separationstr, skiprows=1)
t = data[:, 0]
deer = data[:, 1:]
tmax = max(t)

# Plotting the traces
color = ['k', 'r', 'g', 'b', 'm', 'y']
startingSHC = np.eye(n_trace)
plt.figure(figsize=(7,5))
for i in range(0, n_trace):
   plt.plot(t, deer[:, i] + i)#, color[i])
   plt.title('Traces')
   plt.xlabel('t, ms')
   plt.ylabel('intensity, arb. units')


# %% PRINCIPAL COMPONENT ANALYSIS

# SVD Calculation
u, s, vh = la.svd(deer[:, :])
m = u[:, :n_trace] * s


w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))

# Plots prinicipal components and average values
for i in range(n_trace):
   plt.plot(t[:], i * 1.1 + m[:, i], 'r')
   plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
```

```
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.title('Average Value and \n Principal Components')
plt.xlabel('t, ms')
plt.tight_layout()


# Truncates splined trace dataset based on principal components
s_app = s
smax = pc_used  #the number of principle component curves added to the curves
s_app[smax:n_trace] = 0
m_app = u[:, :n_trace] * s_app
deer_app = np.dot(m_app, vh)
shuffle = np.arange(n_trace)

# Plots the principal components overlaid on top of the DEER traces
w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))
for i in range(n_trace):
   plt.plot(t[:], shuffle[i] * 1.1 + deer_app[:, i], 'k')
   plt.plot(t[:], shuffle[i] * 1.1 + deer[:, i], 'r')

# Axis setup and labelling
ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.ylim([-1,8])
plt.yticks([])
if pc_used == 1:
   plt.title('Traces Overlaid with ' + np.str(smax) + '\n Principal Component')
else:
   plt.title('Traces Overlaid with ' + np.str(smax) + '\n Principal Components')
plt.xlabel('t, ms')
plt.tight_layout()
```

```
# Saves the m matrix to a file along with the time values
pltmatrix = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
   for n in range(n_trace+1):
      if n == 0:
         pltmatrix[i, n] = t[i]
      else:
         pltmatrix[i, n] = m[i, n-1]
np.savetxt(saveStr_pc, pltmatrix, delimiter=separationstr)

# Saves the deer matrix to a file along with the time values
pltmatrix_deer = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
   for n in range(n_trace+1):
      if n == 0:
         pltmatrix_deer[i, n] = t[i]
      else:
         pltmatrix_deer[i, n] = deer_app[i, n-1]
np.savetxt(saveStr_deer_app, pltmatrix_deer, delimiter=separationstr)

# %% CENTERED PCA

# Centres the data and then performs SVD
xmean = np.mean(deer, axis=1)
X = deer.transpose() - xmean.transpose()
X = X.transpose()
u, s, vh = la.svd(X)
m_c = u[:, :n_trace] * s

# Plots the centred principal curves
w, h = figaspect(1.5)
fig = plt.figure(figsize=(w, h))
for i in range(n_trace):
   plt.plot(t[:], i * 1.1 + m_c[:, i], 'r')
   plt.plot(t[:], i * 1.1 + 0 * t[:], 'k', linewidth=0.5)

# Axis setup and labelling
plt.plot(t[:], -1.1 + xmean, 'r')
plt.plot(t[:], -1.1 + 0* t[:], 'k', linewidth=0.5)
plt.title("Average Value and \n Principal Components \n (Centred)")
```

```
plt.xlabel("t, ms")
ax = plt.gca()
ax.xaxis.set_major_locator(MultipleLocator(0.5))
ax.xaxis.set_minor_locator(MultipleLocator(0.1))
ax.tick_params('both', length=10, width=1, which='major')
ax.tick_params('both', length=5, width=1, which='minor')
plt.xlim([0, tmax])
plt.yticks([])
plt.tight_layout()

# Saves the centred m matrix to a file align with the time values
pltmatrix_c = np.zeros((len(t), n_trace+1))
for i in range(0, len(t)):
   for n in range(n_trace+1):
      if n == 0:
         pltmatrix_c[i, n] = t[i]
      else:
         pltmatrix_c[i, n] = m_c[i, n-1]
np.savetxt(saveStr_centre, pltmatrix_c, delimiter=separationstr)
```

MPP Code - Final

09 May 2021 16:04

This is the MPP weight/distance distribution calculation code in its final form.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter, AutoMinorLocator)
from matplotlib.figure import figaspect
import orisel as os
from sklearn import linear_model

# Function to delete rows in data to cut down the number of points to speed up the running of the code
def delete_rows(mat, spacing):
    n_r = len(mat) / spacing
    n_c = len(mat[0])
    new_mat = np.zeros((int(len(mat) - n_r), int(n_c)))

    j = 0
    for i in range(len(mat)):
        if i % spacing != 0 or i == 0:
            for n in range(n_c):
                new_mat[j - 1, n] = mat[i, n]
            j += 1

    return new_mat

### SETUP INFORMATION



# File names and variable separation parameter
fileStr = 'INSERT FILE NAME HERE'
fileStr = fileStr + '.txt'
separationstr = '   '

# Font size configuration
SMALL_SIZE = 10
MEDIUM_SIZE = 16
```

```
BIGGER_SIZE = 24

plt.rc('font', size=SMALL_SIZE)        # controls default text sizes
plt.rc('axes', titlesize=BIGGER_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title

rc('font',**{'family':'serif','serif':['Arial']})



# Number of meaningful principal components
n_bas = 4
# Delete every delete_spacing-th point
delete_spacing = 2
delete_any_points = True



### END OF SETUP INFORMATION



# Loads the data from the file and performs the specified point deletion
data_load = np.loadtxt(fileStr, delimiter=separationstr, skiprows=1)
if delete_any_points == True:
   data = delete_rows(data_load, delete_spacing)
else:
   data = data_load

# Variable setup
t = data[:, 0]
i_deer = data[:, 1:]
tmax = max(t)

n_trace = len(data[0]) - 1

r = np.linspace(1, 8, 150)
f = os.gaussian(r, 3.75, 0.13)

# Plots the traces
```

```
startingSHC = np.eye(n_trace)
plt.figure()
for i in range(0, n_trace):
   plt.plot(t, i_deer[:, i] + i)
   plt.title("Traces")
   plt.xlabel('t, ms')
   plt.ylabel('intensity, arb. units')

deer = np.zeros_like(i_deer)

for i in range(n_trace):
   deer[:, i] = i_deer[:, i]

# %% finding optimal MPP weights
n_time = len(t)
n_r = len(r)

# Spherical harmonic + kernal calculations
basis_kernel = np.zeros((n_bas, n_time, n_r))
i = 0
for v in np.eye(n_bas, n_bas):
   K = os.make_kernel(t, r, v)
   basis_kernel[i, :, :] = K
   i = i + 1

p = startingSHC
kernel = np.zeros((n_trace, n_time, n_r))
full_kernel = np.reshape(np.ravel(kernel), (n_trace * n_time, n_r))
full_trace = np.transpose(deer)
full_trace = full_trace.ravel()

x = f
alpha = 1
# startingSHC
starting_weights = np.eye(n_trace, n_bas)

reg = linear_model.Ridge(alpha=1, fit_intercept=False)

trace = np.zeros_like(deer)
```

```
rnorm_full = []
for k in range(1000):

    A = basis_kernel @ x
    new_weights = np.zeros_like(starting_weights)

    for i in range(n_trace):
        b = deer[:, i]
        reg.fit(A.transpose(), b)
        new_weights[i, :] = reg.coef_

    p = new_weights

    for j in range(n_trace):
        N = np.zeros((n_time, n_r))
        for i in range(n_bas):
            N = N + p[j, i] * basis_kernel[i, :, :]

        kernel[j, :, :] = N
    trace = deer

    full_kernel = np.reshape(np.ravel(kernel), (n_trace * n_time, n_r))
    full_trace = np.transpose(trace)
    full_trace = full_trace.ravel()
    x, rnorm = os.solve_tikhonov(full_kernel, full_trace, alpha)
    print(rnorm)
    rnorm_full.append(rnorm)

# Figure plotting
plt.figure()
plt.plot(full_kernel @ x)
plt.plot(full_trace)
plt.title("Trace/Kernal Plot")
plt.legend(["Kernal", "Trace"])
plt.figure()
plt.plot(r, x)
plt.title("Distance Distribution")
plt.xlabel("Distance/nm")
plt.ylabel("f(r) arb. units")
plt.figure()
plt.plot(np.asarray(rnorm_full))
```

```
plt.title("Convergence of Normalised r")
plt.xlabel("k")
plt.ylabel("Normalised r")

# Calculation of the distance distribution peak
x_max_index = 0
x_max = max(x)
for i in range(len(x)):
   if x[i] == x_max:
      x_max_index = i
print('Distance distribution peak at ' + str(r[x_max_index]) + 'nm.')
```

Orisel Library - Final

09 May 2021 16:09

In order to calculate correct distance distributions we needed to make a small modification to Dr. Potapov's Orisel library, removing a normalisation factor. This is the final orisel library used with our code. The modification we made was in the 'make_kernel' function – a factor of 1e3 was changed to 1, leaving the arbitrary factor of 1 to show where the modification occured. This change has been highlighted in the code below.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 21 17:19:39 2019

@author: alex
"""
import numpy as np
import constants as const
from numpy import linalg as la
import math
from matplotlib import cm, colors
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.colors
import matplotlib.pyplot as plt
import scipy.special as sp
import numpy.fft as ft
import scipy.interpolate as ip
import scipy.signal as sign
import scipy.optimize as opt

def make_kernel(t, r, SHC):
    Kernel = np.zeros((len(t),len(r)))
    K0 = DEER(SHCs = SHC)
    for i in range(len(r)):
        r1 = np.array([r[i]])
        f1 = np.array([1])
        Kernel[:,i] = K0.trace(r1, f1, t/1)
    return Kernel

def solve_tikhonov(Kernel, y, alpha):
    aI = alpha*np.eye(Kernel.shape[1], Kernel.shape[1])
```

```
    K = np.concatenate((Kernel,aI))
    #z = np.zeros(Kernel.shape[1])
    #b = np.concatenate((trace, z))
    b = np.pad(y,(0,Kernel.shape[1]),mode='constant',constant_values=(0,0))
    f, rnorm = opt.nnls(K,b)
    return f, rnorm

def gaussian(x, c, k):
    return np.exp(-np.power((x - c)/k, 2.))

def pump(freq, freq1):
    tp = 1/freq1/4
    freq_eff = np.sqrt(np.power(freq,2) + np.power(freq1, 2))
    scale = np.power( freq1/freq_eff, 2)
    sin2  = np.power( np.sin(freq_eff*tp*2*const.PI)  ,2)
    return scale*sin2

def obs(freq, freq1):
    tp = 1/freq1/4
    freq_eff = np.sqrt(np.power(freq,2) + np.power(freq1, 2))
    scale = np.power( freq1/freq_eff, 5)
    sin2  = np.power( np.sin(freq_eff*tp*2*const.PI), 5)
    return scale*sin2

def integrate_sphere(f, surf):
    #print (np.sqrt( np.power(surf[:,0], 2) + np.power(surf[:,1], 2) ))
    sint = np.sqrt( np.power(surf[:,0], 2) + np.power(surf[:,1], 2) )
    #print ("now sint")
    #print (sint)
    return f@sint

def integrate_SHC_product(coeff1, coeff2):
    c1 = coeff1[0,:,:]
    c2 = coeff2[0,:,:]
    s1 = coeff1[1,:,:]
    s2 = coeff2[1,:,:]
    return np.sum(c1*c2) - np.sum(s1*s2)


def plot_sphere(f, surf, nK, fmax, fmin):
```

```
    # PLOT THE EXCITATION PROFILE ON A SPHERE
    fcolors = f.real
    fcolors = fcolors.reshape((nK,nK))
    #fmax, fmin = fcolors.max(), fcolors.min()
    #fmax = 1
    #fmin = 0
    fcolors = (fcolors - fmin)/(fmax - fmin)

    # Set the aspect ratio to 1 so our sphere looks spherical
    norm = matplotlib.colors.Normalize(fmin, fmax)

    fig = plt.figure(figsize=plt.figaspect(1.0))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(surf[:,0].reshape((nK,nK)), surf[:,1].reshape((nK,nK)), surf[:,2].reshape(nK,nK),
rstride=1, cstride=1, facecolors=cm.jet(norm(fcolors)))
    #ax.plot_surface(surf[:,0].reshape((nK,nK)), surf[:,1].reshape((nK,nK)), surf[:,2].reshape(nK,nK),
rstride=1, cstride=1, facecolors=cm.seismic(norm(fcolors)))



    ax.view_init(30,125)

    m = cm.ScalarMappable(cmap=plt.cm.jet, norm=norm)
    #m = cm.ScalarMappable(cmap=plt.cm.seismic, norm=norm)
    m.set_array([])
    #plt.colorbar(m)

    # Turn off the axis planes
    ax.set_axis_off()
    plt.show()
    return fig



class SpinSystem:
    """
      The definition of spin system parameters. Currently works for
      S = 1/2 and I = 1. In principle can be extended for more diverse
      nuclei and a bigger number of them.

      The class methods:
        get_hamiltonian(B, directional_cosines)  - calculates the Hamiltonian
                                        operator matrix
```

```
    eigen_freqs(B, directional_cosines)     - eigen frequencies and
                                    eigenvectors
    transitions((B, directional_cosines))   - transition frequencies and
                                    their intensities
"""
def __init__(self, **kwargs):
    self.spin = '1/2'
    self.nuclei = '14N'
    self.g_tensor = [2.002, 2.000, 2.000]
    self.hyperfine = [10, 10, 20]

    for key, value in kwargs.items():
        if key =='Spin':
            self.spin = value
        if key =='Nuclei':
            self.nuclei = value
        if key =='G':
            self.g_tensor = value
        if key =='A':
            self.hyperfine = value

    if const.N_SPIN[self.nuclei] == '1/2':
        nucl_dim = 2
    if const.N_SPIN[self.nuclei] == '1':
        nucl_dim = 3
    if self.spin == '1/2':
        elec_dim = 2
    total_dim = nucl_dim * elec_dim


    S_x = np.array([0, 0.5 , 0.5, 0])*(1.0+0.0j)
    S_y = np.array([0, -0.5j, 0.5j, 0])
    S_z = np.array([0.5, 0 , 0, -0.5])*(1.0+0.0j)
    S_0 = np.array([1, 0, 0, 1])
    Sop = [S_0, S_x, S_y, S_z]


    I_z = np.array([1, 0, 0 , 0, 0, 0 , 0, 0, -1])*(1.0+0.0j)
    I_y = np.array([0, 1, 0, -1, 0, 1 , 0, -1, 0])/np.sqrt(2)/(0.0+1.0j)
    I_x = np.array([0, 1, 0 , 1, 0, 1 , 0, 1, 0 ])/np.sqrt(2)
```

```
    I_0 = np.array([ 1, 0, 0 , 0, 1, 0 , 0, 0, 1])*(1.0+0.0j)
    Iop = [I_0, I_x, I_y, I_z]

    SIop = np.zeros((4, 4, len(S_0)*len(I_0)))*(1.0+0.0j)

    self.lenS = len(S_0)
    self.lenI = len(I_0)
    for ix, S in enumerate(Sop):
        for jx, I in enumerate(Iop):
            temp = np.matmul(S.reshape(self.lenS,1),I.reshape(1,self.lenI))
            SI = temp.reshape(1, self.lenS*self.lenI)
            SI = SI.squeeze()
            SIop[ix][jx] = SI
    self.SIop = np.zeros((4, 4, total_dim, total_dim))*(1.0+0.0j)
    for ix in range(4):
        for jx in range(4):
            temp = SIop[ix][jx].reshape(self.lenS, self.lenI)
            M = np.block([[temp[0].reshape(nucl_dim,nucl_dim),
                           temp[1].reshape(nucl_dim,nucl_dim)],
                          [temp[2].reshape(nucl_dim,nucl_dim),
                           temp[3].reshape(nucl_dim,nucl_dim)]])
            self.SIop[ix][jx] = M

def __get_matrix(self, A):
    # this would work only for S=1/2 system
    temp = A.reshape(self.lenS, self.lenI)
    n_dim = int(np.round(np.sqrt(self.lenI)))

    M = np.block([ [temp[0].reshape(n_dim,n_dim), temp[1].reshape(n_dim,n_dim)],
                   [temp[2].reshape(n_dim,n_dim), temp[3].reshape(n_dim,n_dim)] ])
    return M

def get_hamiltonian(self, B, directional_cosines):

    b_x = B*directional_cosines[0]
    b_y = B*directional_cosines[1]
    b_z = B*directional_cosines[2]
    A = self.hyperfine
    G = self.g_tensor

    H_ez = (b_x*G[0]*self.SIop[1][0] +
```

```
            b_y*G[1]*self.SIop[2][0] +
            b_z*G[2]*self.SIop[3][0])*const.GAMMA_E/const.G_E/(2*const.PI)/1e6
        H_hyp= (A[0]*self.SIop[1][1]  +
            A[1]*self.SIop[2][2] +
            A[2]*self.SIop[3][3])
        H_nz = (b_x*self.SIop[0][1] +
            b_y*self.SIop[0][2] +
            b_z*self.SIop[0][3])*const.GAMMA_N['14N']/(2*const.PI)/1e6

        H = H_ez + H_hyp + H_nz

        return H

    def eigen_freqs(self, B, directional_cosines):
        H = self.get_hamiltonian(B, directional_cosines)
        evals, evecs = la.eig(H)
        return evals, evecs

    def transitions(self, B, directional_cosines):
        """
        Calculates transition frequencies and transition
        probabilities for of hamiltonian for a given B and its direction
        """
        evals, evecs = self.eigen_freqs(B, directional_cosines)
        evecs_c = np.conj(evecs)
        prob = []

        """
         In order to find the transition probability, one has to find first
         the directions of an oscillating field perpendicular to the main B_z
         In fact the direction of B_1 in the x-y plane is not important for
         the value of the matrix element
         Therefore any vector perpedicular to B can be used for the
         direction of the oscillating field
        """
        v1 = directional_cosines[0:3]
        if abs(v1[1]) <1e-6:
            v2 = [0, 1, 0]
        else:
            n1 = math.sqrt(1/(1+ (v1[0]/v1[1])**2))
```

```
            n2 = -n1*v1[0]/v1[1]
            v2 = np.array([n1, n2, 0])/math.sqrt(n1**2 + n2**2)
        B1_direction = v2[0]*self.SIop[1][0]+v2[1]*self.SIop[2][0]+v2[2]*self.SIop[3][0]
        """
         la.eig has a rather non-intuitive output where the eigenvectors
         are placed as COLUMNs in the evecs output matrix, for that reason
         iterating over all the eigenvectors requires transposing the
         evecs matrix
        """
        for vec1 in evecs.T:
            for vec2 in evecs_c.T:
                matrix_element = vec2@(B1_direction@vec1)
                prob.append(matrix_element*np.conj(matrix_element))
        freq = []
        for val1 in evals:
            for val2 in evals:
                freq.append(np.abs(val1 - val2))
        """
         The normalization factor of 3/2 comes from:
            three values of I projection
            another factor of 2 - not quite clear
        """
        return np.array(freq), np.array(prob)*2/3


class SpinSolve:
    """
     Contains the experimental parameters for calculating the spectra
     Currently contains only one method:
       epr_spectrum() which calcalutes the first order EPR spectrum
                  (0th order derivative)
    """
    def __init__(self, spin_system, **kwargs):
        # SET UP THE DEFAULT PARAMETERS FOR THE CALCULATION
        self.mw_freq = 94900
        self.exc_band = 10
        self.n_points = 100
        self.spin_system = spin_system
        self.field = [3.37, 3.39]
        self.grid = 'DH'
        self.n_knots = 0
```

```
self.pulse ='gaussian'
n_knots = 0

# READ THE ARGUMENTS TO PROVIDE PARAMETERS FOR THE CALCULATION

for key, value in kwargs.items():
   if key =='MWFreq':
      self.mw_freq = value
   if key =='ExcitationBandwidth':
      self.exc_band = value
   if key =='Pulse':
      self.pulse = value
   if key =='NPoints':
      self.n_points = value
   if key =='MagneticField':
      self.field = value
   if key == 'Grid':
      self.grid = value
   if key == 'nKnots':
      self.n_knots = value
      n_knots = self.n_knots


if self.grid!='DH2' and self.grid!='DH':
   print("unsupported grid, defaulting to DH grid")
   self.grid = 'DH'
if self.grid == 'DH2':
   if self.n_knots == 0:
      print("default number of nKnots = 10")
      n_knots = 10
      self.n_knots = n_knots
   phiv = np.linspace(0, 2*const.PI*(2*n_knots - 1)/(2*n_knots),
                  2*n_knots)
   thetav = np.linspace(0, const.PI*(n_knots - 1)/(n_knots),
                    n_knots)
   self.grid = np.meshgrid(phiv, thetav)

if self.grid == 'DH':
   if self.n_knots == 0:
      print("default number of nKnots = 10")
```

```
            n_knots = 10
            self.n_knots = n_knots
        phiv = np.linspace(0, 2*const.PI*(n_knots - 1)/(n_knots),
                           n_knots)
        thetav = np.linspace(0, const.PI*(n_knots - 1)/(n_knots),
                             n_knots)
        self.grid = np.meshgrid(phiv, thetav)


    if isinstance(self.field, list):
        self.field_range =  np.linspace(self.field[0], self.field[1],
                                self.n_points)


    # INITIALIZE THE DIRECTIONAL COSINES FOR A GIVEN GRID
    phiv = self.grid[0]
    thetav = self.grid[1]

    x = np.sin(thetav)*np.cos(phiv)
    y = np.sin(thetav)*np.sin(phiv)
    z = np.cos(thetav)
    w = np.sin(thetav)   # weights

    x = x.reshape(1, x.size)
    y = y.reshape(1, y.size)
    z = z.reshape(1, z.size)
    w = w.reshape(1, w.size)


    directional_cosines = np.block([[x],[y],[z],[w]])
    self.directional_cosines = directional_cosines.T

def epr_spectrum(self):
    #print(self.spin_system.hyperfine)
    Spec = []
    C = math.sqrt(math.log(2))
    #print (self.directional_cosines)
    # the weights of various orientations are not taken into account yet
    for B in self.field_range:
        s = []
        for dir_cos in self.directional_cosines:
```

```
            freq, prob= self.spin_system.transitions(B, dir_cos[0:3])
            if self.pulse == 'pump':
               t = prob*pump(freq-self.mw_freq, self.exc_band)
               #t = prob*gaussian(freq, self.mw_freq, self.exc_band)
            elif self.pulse == 'obs':
               t = prob*obs(freq-self.mw_freq, self.exc_band)
            else:
               t = prob*gaussian(freq, self.mw_freq, self.exc_band/C)

            s.append(np.sum(t))
         s = np.array(s)
         Spec.append(np.sum(s*self.directional_cosines[:,3]))

      return self.field_range, np.array(Spec)

   def excite_sphere(self):
      B = self.field
      sph_excitation_profile = []
      C = math.sqrt(math.log(2))
      for dir_cos in self.directional_cosines:
         freq, prob= self.spin_system.transitions(B, dir_cos[0:3])
         #t = prob*gaussian(freq, self.mw_freq, self.exc_band)
         if self.pulse == 'pump':
            t = prob*pump(freq-self.mw_freq, self.exc_band)
         elif self.pulse == 'obs':
            t = prob*obs(freq-self.mw_freq, self.exc_band)
         else:
            t = prob*gaussian(freq, self.mw_freq, self.exc_band/C)

         sph_excitation_profile.append(np.sum(t))
      #return np.array(sph_excitation_profile)/self.directional_cosines[:,3],
self.directional_cosines[:,0:3]
      return np.array(sph_excitation_profile), self.directional_cosines[:,0:3]

def SH_full_alpha_filter(coeff, lmax):
   temp = np.zeros_like(coeff)
   # averaging over ALPHA angle
   for i in range(2):
      for j in range(lmax+1):
         temp[i,j,0] = coeff[i,j,0]
```

```
    return temp

def SH_full_gamma_filter(coeff, lmax):
    return SH_full_alpha_filter(coeff, lmax)

def SH_beta_filter(beta, beta_width, djpi2, coeff, lmax):
    Sb = beta_width
    Beta = beta

    M = np.linspace(0, lmax,lmax +1)

    cosmpi2 = np.cos(M*np.pi/2)
    sinmpi2 = np.sin(M*np.pi/2)
    cosmb = np.cos(M*Beta)
    sinmb = np.sin(M*Beta)

    A = np.ones(2*lmax+2)
    A[1:(2*lmax+2):2] = -1

    a1 = np.ones((lmax+1,lmax+1))
    a2 = np.ones((lmax+1,lmax+1))

    a1[:,1:(lmax+1):2] = -1
    a2[1:(lmax+1):2,:] = -1

    p = np.zeros_like(djpi2)
    q = np.zeros_like(djpi2)
    r = np.zeros_like(djpi2)
    s = np.zeros_like(djpi2)
    #print(a2)

    for j in range(lmax+1):
        #print (j)
        if (j % 2) == 0:
            sign = 1
        else:
            sign = -1
        p[j,:,:] =  a1 + sign*a2
        q[j,:,:] =  a1 - sign*a2
        r[j,:,:] =  a2 + sign*a1
        s[j,:,:] =  a2 + sign*a1
```

```
t1 = np.swapaxes(djpi2,0,2)
t = np.swapaxes(t1,1,2)

p1 = p*t
q1 = q*t
r1 = r*t
s1 = s*t

#temp = np.zeros_like(coeff)
temp = coeff
# rotate by pi/2 around Z
for i in range(lmax+1):
   temp[0,i,:] =  temp[0,i,:]*cosmpi2 + temp[1,i,:]*sinmpi2
   temp[1,i,:] = -temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2

#rotate by pi/2 around Y
I = np.ones(lmax+1)
I[1:lmax+1:2] = -1
for i in range(lmax+1):
   temp[0,i,:] = I*p1[i,:,:]@temp[0,i,:] # C_lm coefficient
   temp[1,i,:] = I*q1[i,:,:]@temp[1,i,:] # S_lm coefficient

#rotate by BETA angle
for i in range(lmax+1):
   temp[0,i,:] =  temp[0,i,:]*cosmb + temp[1,i,:]*sinmb
   temp[1,i,:] = -temp[0,i,:]*sinmb + temp[1,i,:]*cosmb

# averaging over BETA angle
scale_b = gaussian(M*Sb/2, 0, 1)
#print (scale_a)
for i in range(2):
   for j in range(lmax+1):
      temp[i,j,:] = temp[i,j,:]*scale_b

#temp = temp/np.sum(scale_b)

#rotate by -pi/2 around Y
for i in range(lmax+1):
   temp[0,i,:] = I*r1[i,:,:]@temp[0,i,:] # C_lm coefficient
```

```
        temp[1,i,:] = I*s1[i,:,:]@temp[1,i,:] # S_lm coefficient

    # rotate by -pi/2 around Z
    for i in range(lmax+1):
        temp[0,i,:] =  temp[0,i,:]*cosmpi2 - temp[1,i,:]*sinmpi2 # C_lm coefficient
        temp[1,i,:] =  temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2 # S_lm coefficient



    """
    I need to add nulling of the Sin coefficients for m = 0
    """
    return temp

def SH_gaussian_alpha_filter(alpha, width, coeff, lmax):
    Sa = width
    M = np.linspace(0, lmax,lmax +1)
#   cosmpi2 = np.cos(M*np.pi/2)
#   sinmpi2 = np.sin(M*np.pi/2)
    A = np.ones(2*lmax+2)
    A[1:(2*lmax+2):2] = -1
    cosma = np.cos(M*alpha)
    sinma = np.sin(M*alpha)

    temp = coeff
    # rotate over ALPHA angle
    for i in range(lmax+1):
        temp[0,i,:] =  temp[0,i,:]*cosma + temp[1,i,:]*sinma
        temp[1,i,:] = -temp[0,i,:]*sinma + temp[1,i,:]*cosma

    #averaging over ALPHA angle
    scale_a = gaussian(M*Sa/2, 0, 1)
    #print (scale_a)
    for i in range(2):
        for j in range(lmax+1):
            temp[i,j,:] = temp[i,j,:]*scale_a
    #temp = temp/np.sum(scale_a)
    return temp

def SHGaussianFilter(width, djpi2, coeff, lmax):
    Sa = width[0]
    Sb = width[1]
```

```
Sg = width[2]

M = np.linspace(0, lmax,lmax +1)
Mi = M.astype(int)

cosmpi2 = np.cos(M*np.pi/2)
sinmpi2 = np.sin(M*np.pi/2)
A = np.ones(2*lmax+2)
A[1:(2*lmax+2):2] = -1

a1 = np.ones((lmax+1,lmax+1))
a2 = np.ones((lmax+1,lmax+1))

a1[:,1:(lmax+1):2] = -1
a2[1:(lmax+1):2,:] = -1

p = np.zeros_like(djpi2)
q = np.zeros_like(djpi2)
r = np.zeros_like(djpi2)
s = np.zeros_like(djpi2)
#print(a2)

for j in range(lmax+1):
   #print (j)
   if (j % 2) == 0:
      sign = 1
   else:
      sign = -1
   p[j,:,:] =  a1 + sign*a2
   q[j,:,:] =  a1 - sign*a2
   r[j,:,:] =  a2 + sign*a1
   s[j,:,:] =  a2 + sign*a1

t1 = np.swapaxes(djpi2,0,2)
t = np.swapaxes(t1,1,2)

p1 = p*t
q1 = q*t
r1 = r*t
s1 = s*t
```

```
temp = coeff[:,:(lmax+1),:(lmax+1)]

# averaging over ALPHA angle
scale_a = gaussian(M*Sa/2, 0, 1)
#print (scale_a)
for i in range(2):
   for j in range(lmax+1):
      temp[i,j,:] = temp[i,j,:]*scale_a

# rotate by pi/2 around Z
for i in range(lmax+1):
   temp[0,i,:] =  temp[0,i,:]*cosmpi2 + temp[1,i,:]*sinmpi2
   temp[1,i,:] = -temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2

#rotate by pi/2 around X
I = np.ones(lmax+1)
I[1:lmax+1:2] = -1
for i in range(lmax+1):
   temp[0,i,:] = I*p1[i,:,:]@temp[0,i,:] # C_lm coefficient
   temp[1,i,:] = I*q1[i,:,:]@temp[1,i,:] # S_lm coefficient

# averaging over BETA angle

# produce an array with Fourier coeffiecients for abs(sin beta)
N1 = np.linspace(0,2*lmax,2*lmax+1)
b = np.zeros(len(N1))
b[0] = 2/math.pi
b[1] = 0
b[2:(2*lmax+1)] = -2/math.pi*(1 + (-1)**N1[2:(2*lmax+1)])/(np.power(N1[2:(2*lmax+1)],2) -1)

# Test that Fourier Series coefficients are correct
#x = np.linspace(-math.pi, math.pi, 100)
#s = np.zeros_like(x)
#for n in range(lmax+1):
#    s = s + b[n]*np.cos(n*x)

# calculate scaling factor
N2 = np.linspace(-(2*lmax),2*lmax+1, 4*lmax+1)
expn = 1/2*gaussian(N2*Sb/2, 0, 1)
scale_b = np.zeros_like(M)
```

```
    for m in Mi:
        expnp = expn[ m+2*lmax: (m+3*lmax+1)]
        expnm = expn[-m+2*lmax:(-m+3*lmax+1)]
        scale_b[m] = b[0:(lmax+1)]@(expnp+expnm)
    scale_b = scale_b/scale_b[0]
    # now apply scaling due to BETA averaging
    for i in range(2):
        for j in range(lmax+1):
            temp[i,j,:] = temp[i,j,:]*scale_b

    #rotate by -pi/2 around X
    for i in range(lmax+1):
        temp[0,i,:] = I*r1[i,:,:]@temp[0,i,:] # C_lm coefficient
        temp[1,i,:] = I*s1[i,:,:]@temp[1,i,:] # S_lm coefficient

    # rotate by -pi/2 around Z
    for i in range(lmax+1):
        temp[0,i,:] =  temp[0,i,:]*cosmpi2 - temp[1,i,:]*sinmpi2 # C_lm coefficient
        temp[1,i,:] =  temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2 # S_lm coefficient

    # averaging over GAMMA angle
    scale_g = gaussian(M*Sg/2, 0, 1)
    #print (scale_g)
    for i in range(2):
        for j in range(lmax+1):
            temp[i,j,:] = temp[i,j,:]*scale_g
    """
    I need to add nulling of the Sin coefficients for m = 0
    """

    return temp



def SHCylindricalFilter(beta, beta_width, djpi2, coeff, lmax):
    Sb = beta_width
    Beta = beta

    M = np.linspace(0, lmax,lmax +1)
    Mi = M.astype(int)
```

```
cosmpi2 = np.cos(M*np.pi/2)
sinmpi2 = np.sin(M*np.pi/2)
cosmb = np.cos(M*Beta)
sinmb = np.sin(M*Beta)

A = np.ones(2*lmax+2)
A[1:(2*lmax+2):2] = -1

a1 = np.ones((lmax+1,lmax+1))
a2 = np.ones((lmax+1,lmax+1))

a1[:,1:(lmax+1):2] = -1
a2[1:(lmax+1):2,:] = -1

p = np.zeros_like(djpi2)
q = np.zeros_like(djpi2)
r = np.zeros_like(djpi2)
s = np.zeros_like(djpi2)
#print(a2)

for j in range(lmax+1):
   #print (j)
   if (j % 2) == 0:
      sign = 1
   else:
      sign = -1
   p[j,:,:] =  a1 + sign*a2
   q[j,:,:] =  a1 - sign*a2
   r[j,:,:] =  a2 + sign*a1
   s[j,:,:] =  a2 + sign*a1

t1 = np.swapaxes(djpi2,0,2)
t = np.swapaxes(t1,1,2)

p1 = p*t
q1 = q*t
r1 = r*t
s1 = s*t

temp = np.zeros_like(coeff)
```

```
# averaging over ALPHA angle
 for i in range(2):
    for j in range(lmax+1):
      temp[i,j,0] = coeff[i,j,0]

 # rotate by pi/2 around Z
 for i in range(lmax+1):
    temp[0,i,:] =  temp[0,i,:]*cosmpi2 + temp[1,i,:]*sinmpi2
    temp[1,i,:] = -temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2

#rotate by pi/2 around X
 I = np.ones(lmax+1)
 I[1:lmax+1:2] = -1
 for i in range(lmax+1):
    temp[0,i,:] = I*p1[i,:,:]@temp[0,i,:] # C_lm coefficient
    temp[1,i,:] = I*q1[i,:,:]@temp[1,i,:] # S_lm coefficient

 #rotate by BETA angle
 for i in range(lmax+1):
    temp[0,i,:] =  temp[0,i,:]*cosmb + temp[1,i,:]*sinmb
    temp[1,i,:] = -temp[0,i,:]*sinmb + temp[1,i,:]*cosmb

 # averaging over BETA angle
 scale_b = gaussian(M*Sb/2, 0, 1)
 #print (scale_a)
 for i in range(2):
    for j in range(lmax+1):
      temp[i,j,:] = temp[i,j,:]*scale_b


 """
  this is another version of averaging procedure
  where the sin(beta) factor is taken into account -
  which is actually not needed
 # averaging over BETA angle

 # produce an array with Fourier coeffiecients for abs(sin beta)
 N1 = np.linspace(0,2*lmax,2*lmax+1)
 b = np.zeros(len(N1))
```

```
b[0] = 2/math.pi
b[1] = 0
b[2:(2*lmax+1)] = -2/math.pi*(1 + (-1)**N1[2:(2*lmax+1)])/(np.power(N1[2:(2*lmax+1)],2) -1)

# Test that Fourier Series coefficients are correct
#x = np.linspace(-math.pi, math.pi, 100)
#s = np.zeros_like(x)
#for n in range(lmax+1):
#    s = s + b[n]*np.cos(n*x)

# calculate scaling factor
N2 = np.linspace(-(2*lmax),2*lmax+1, 4*lmax+1)
expn = 1/2*gaussian(N2*Sb/2, 0, 1)
scale_b = np.zeros_like(M)
for m in Mi:
   expnp = expn[ m+2*lmax: (m+3*lmax+1)]
   expnm = expn[-m+2*lmax:(-m+3*lmax+1)]
   scale_b[m] = b[0:(lmax+1)]@(expnp+expnm)
scale_b = scale_b/scale_b[0]
# now apply scaling due to BETA averaging
for i in range(2):
   for j in range(lmax+1):
      temp[i,j,:] = temp[i,j,:]*scale_b
"""

#rotate by -pi/2 around X
for i in range(lmax+1):
   temp[0,i,:] = I*r1[i,:,:]@temp[0,i,:] # C_lm coefficient
   temp[1,i,:] = I*s1[i,:,:]@temp[1,i,:] # S_lm coefficient

# rotate by -pi/2 around Z
for i in range(lmax+1):
   temp[0,i,:] =  temp[0,i,:]*cosmpi2 - temp[1,i,:]*sinmpi2 # C_lm coefficient
   temp[1,i,:] =  temp[0,i,:]*sinmpi2 + temp[1,i,:]*cosmpi2 # S_lm coefficient

# averaging over GAMMA angle
temp2 = np.zeros_like(temp)
for i in range(2):
   for j in range(lmax+1):
      temp2[i,j,0] = temp[i,j,0]
```

```
    """
    I need to add nulling of the Sin coefficients for m = 0
    """
    return temp2


class ModelDEER:
    def __init__(self, **kwargs):
        self.degree = 8
        self.l = 10000
        omega_max = 10
        dt = 1/(2*omega_max)
        self.omega = np.linspace(-omega_max,omega_max,self.l)
        self.time = np.linspace(0, dt*self.l,self.l)
        self.gauss = gaussian(self.omega, 0, 0.01)
        for key, value in kwargs.items():
            if key =='Degree':
                self.degree = value

    def dipolar_spectrum(self, f, order):
        Spec = np.zeros_like(f)
        I1 = np.argwhere(f<=-2)
        I2 = np.argwhere(np.logical_and(f>-2, f<-1))
        I3 = np.argwhere(np.logical_and(f>=-1, f<=1))
        I4 = np.argwhere(np.logical_and(f>1, f<2))
        I5 = np.argwhere(f>=2)

        Spec[I1] = 0

        z1 = np.sqrt( (1 - f[I2])/3);
        t1 = np.arccos(np.sqrt(z1))
        Spec[I2] = 1/z1 * np.real(sp.sph_harm(0,self.degree,0,t1))

        z1 = np.sqrt( (1 - f[I3])/3);
        t1 = np.arccos(np.sqrt(z1))
        z2 = np.sqrt( (1 + f[I3])/3);
        t2 = np.arccos(np.sqrt(z2))
        Spec[I3] = 1/z1 * np.real(sp.sph_harm(0,self.degree,0,t1)) + \
                1/z2 * np.real(sp.sph_harm(0,self.degree,0,t2))

        z1 = np.sqrt( (1 + f[I4])/3);
```

```
        t1 = np.arccos(np.sqrt(z1))
        Spec[I4] = 1/z1 * np.real(sp.sph_harm(0,self.degree,0,t1))

        Spec[I5] = 0
        return Spec

    def trace(self):
        Spec = self.dipolar_spectrum(self.omega,self.degree)
        Spec_conv = sign.convolve(Spec, self.gauss, mode ='same')
        Spec_conv = np.roll(Spec_conv,self.l//2)
        trace = np.real(ft.ifft(Spec_conv))
        return ip.interp1d(self.time,trace,kind = 'cubic')

class DEER:
    def __init__(self,**kwargs):
        self.SHCs = np.array([1])
        for key, value in kwargs.items():
            if key =='SHCs':
                self.SHCs = value
        self.all_traces = []
        for i in range(len(self.SHCs)):
            m = ModelDEER(Degree = i*2)
            self.all_traces.append(m.trace())
    def model(self, time):
        sum1 = 0
        for i in range(len(self.SHCs)):
            sum1 = sum1 + self.all_traces[i](time)*self.SHCs[i]
        return sum1

    def trace(self, r, f, time):
        freq_dd = 52.16/np.power(r,3)
        sum1 = np.zeros_like(time)
        for j in range(len(self.SHCs)):
            model = self.all_traces[j]
            for i in range(len(freq_dd)):
                sum1 = sum1 + f[i]*model(freq_dd[i]*time)*self.SHCs[j]  #
        return sum1
```