suspect Documentation

Release 0.4.4

Ben Rowland

Apr 16, 2023

CONTENTS

1 Getting started 3

2 Solving specific problems 27

3 The Consensus Processing Pipeline 31

4 API Reference 41

Python Module Index 49

Index 51

Welcome! This is the documentation for Suspect 0.4.4, last updated Apr 16, 2023

Parts of the documentation:

CHAPTER

ONE

GETTING STARTED

1.1 1. Introduction to MRS processing with Suspect

Welcome to the first in our series of tutorials in how to use Suspect for working with MR spectroscopy data. In this introduction, we will present some of the basic features of Suspect while in subsequent tutorials we will explore various parts of the processing pipeline in more detail.

The first thing is to import the `suspect` package. We will also import the `pyplot` module from Matplotlib, which we will use for plotting our data, and we use some Jupyter "magic" to cause all plots to be rendered inline with the text in the notebook.

[ ]:
```
import suspect
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

Let's get started by loading some MRS data. Suspect currently supports various different data formats including Siemens DICOM, RDA and TWIX and Philips SDAT. GE P files are not yet supported but should be soon. The Suspect repository on Github contains a number of example data files and we will be using those throughout this tutorial series, so you can follow along if you like. If you are using the OpenMRSLab Docker image (which we use to write all these tutorials) then you will find this sample data in the folder at `/home/jovyan/suspect/tests/test_data/`. We are going to start with an RDA file for a short echo time single voxel PRESS sequence.

[2]:
```
data = suspect.io.load_rda("/home/jovyan/suspect/tests/test_data/siemens/SVS_30.rda")
print(data)
```

```
<MRSData instance f0=123.234655MHz TE=30.0ms dt=0.833ms>
```

When we print the data object we just loaded we see that it is an instance of the `MRSData` class, and we also get some other information about the acquisition sequence such as the field strength and echo time. The `MRSData` class stores time domain spectroscopy data and is at the core of the Suspect package. It is a subclass of `numpy.ndarray` which means it has all the standard properties, as well as additional MRS specific ones. To learn more about the `MRSData` class and all its functions and properties, check out the *MRSData API Reference*

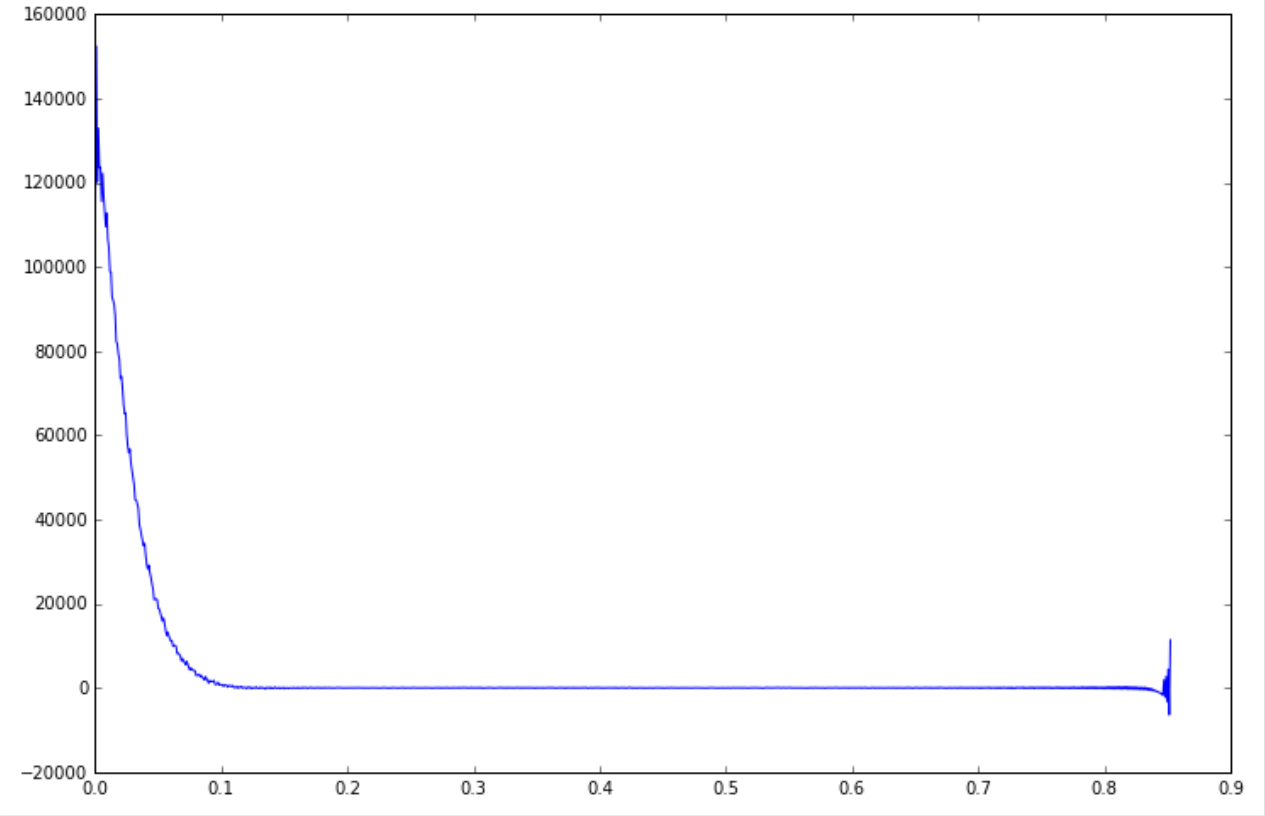
[3]:
```
print(data.shape)
print(data.f0)
print(data.dt)
print(data.sw)
```

```
(1024,)
123.234655
0.000833
1200.4801920768307
```

Now that we have some data loaded, we want to plot it to see what it looks like. As an `ndarray` subclass, `MRSData` is fully compatible with the standard Matplotlib plotting library so displaying our FID is as simple as:

[4]:
```
plt.plot(data.time_axis(), data.real)
```

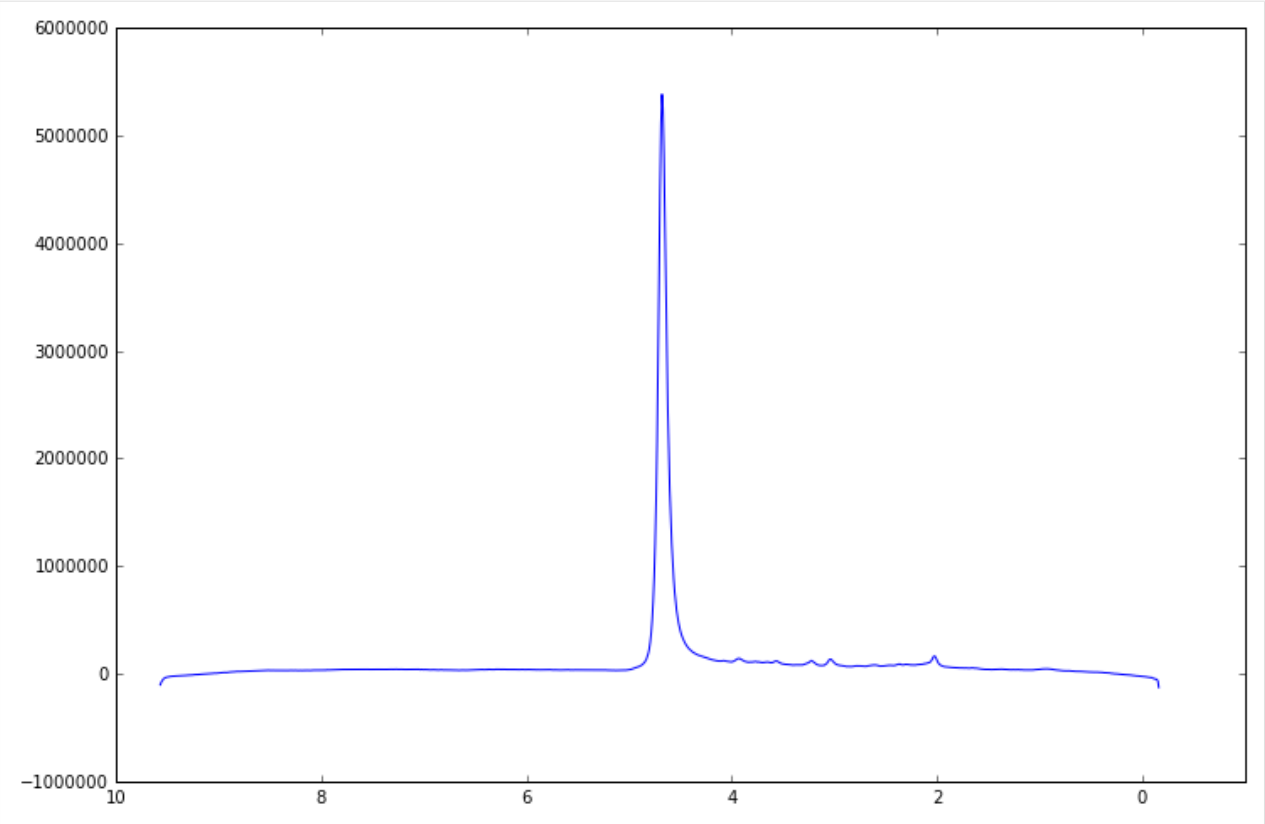
[4]:
```
[<matplotlib.lines.Line2D at 0x7febc9372320>]
```



Note that the `MRSData` object has a `time_axis()` method which gives us the time at which each FID point was acquired. There are similar methods in the frequency domain to give us the axis in Hz or PPM. To obtain the spectrum from the FID we can apply the Fourier transform directly, or just use the helper `spectrum()` method provided by `MRSData`.

[5]:
```
plt.plot(data.frequency_axis_ppm(), data.spectrum().real)
# reverse the limits on the x-axis so that ppm runs from right to left
plt.xlim([10, -1])
```

[5]:
```
(10, -1)
```

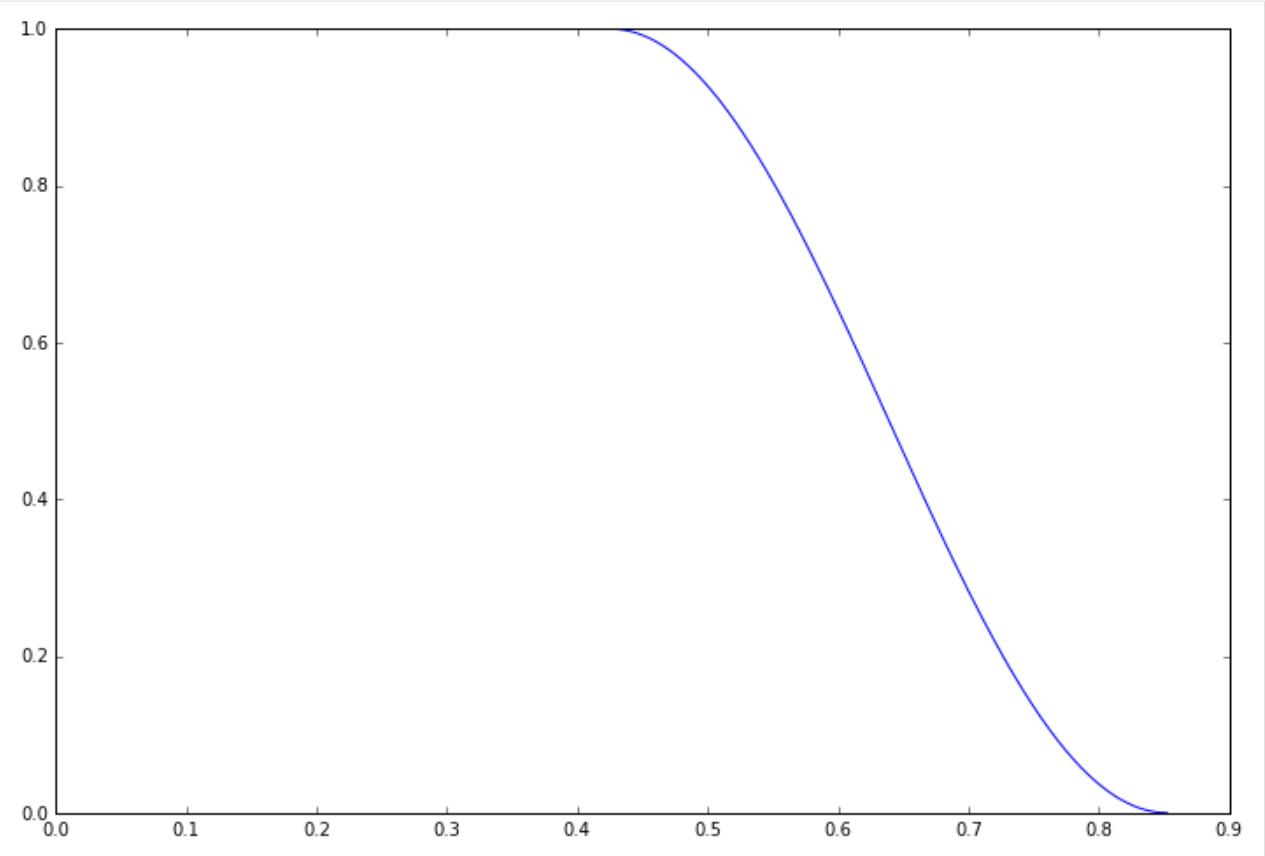


Looking back at the time domain plot we can see that there is a spurious signal which appears at the end of the acquisition window. This kind of artefact can distort the spectrum and cause problems in subsequent processing. For example the end of the FID is often considered to be pure noise, which is used in assessing SNR and these false signals can affect this analysis.

Fortunately we can get rid of this artefact by apodising our data, that is applying a windowing function in the time domain. The `scipy.signal` package provides a huge array of different windowing functions, in our case we will be using the Tukey window as this allows us to leave the first half of the signal completely unaffected and only suppress the signal close to the artefact region.

[6]:
```
import scipy.signal
# scipy.signal produces symmetric windows so only take the second half
window = scipy.signal.tukey(data.np * 2)[data.np:]
plt.plot(data.time_axis(), window)
```

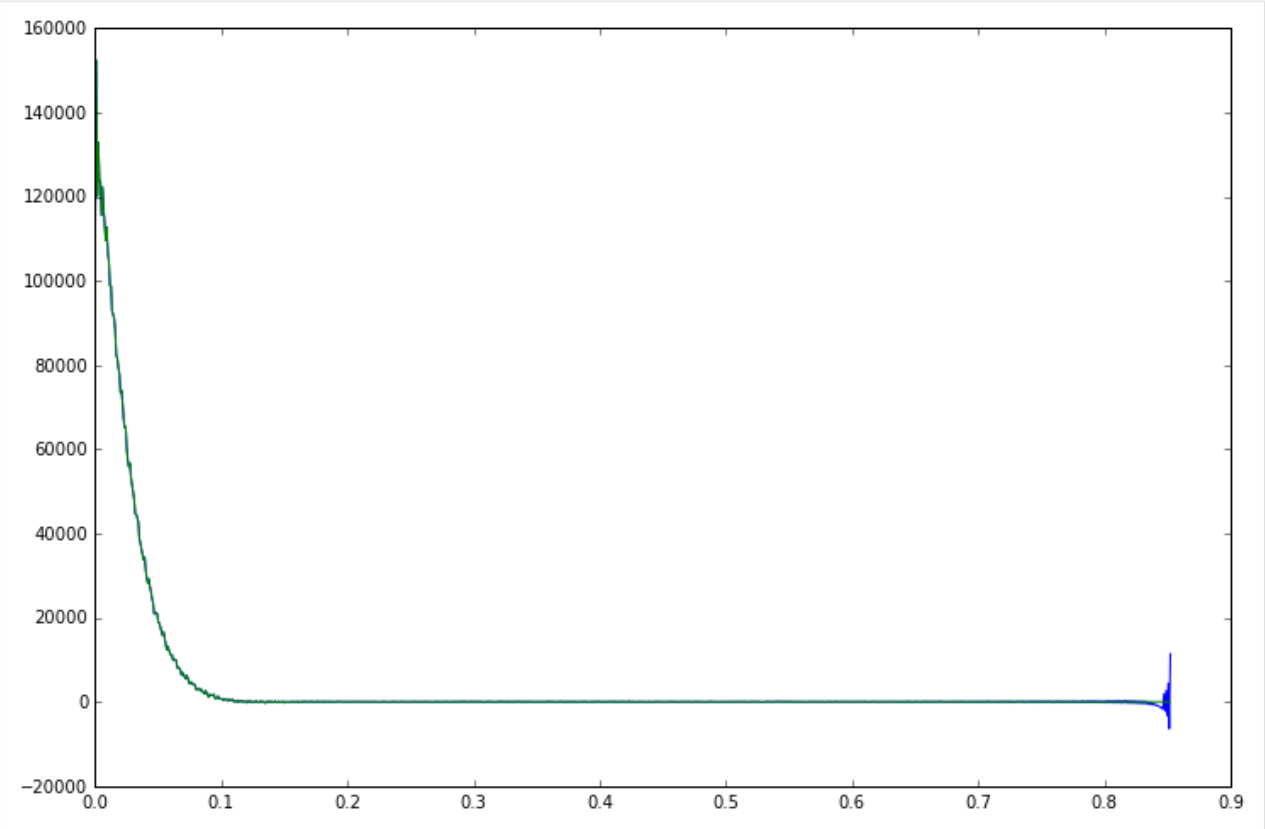
[6]:
```
[<matplotlib.lines.Line2D at 0x7febc91a5a20>]
```



Applying the window to our FID data is as simple as multiplying the two objects together. NumPy will multiply the two arrays componentwise and return the result as a new `MRSData` object, copying all the parameters from the original `data` object, so that the new object keeps all the essential MRS parameters and methods.

[7]:
```
windowed_data = window * data
plt.plot(data.time_axis(), data.real)
plt.plot(windowed_data.time_axis(), windowed_data.real)
```

[7]:
```
[<matplotlib.lines.Line2D at 0x7febc91b97f0>]
```



So that concludes our first look at the Suspect library. We have learnt about the `MRSData` class, how to load MRS data from a file and plot it, in the time and frequency domains, and seen a simple example of removing an artefact through apodisation. In the following tutorials, we will learn how to use Suspect to do more advanced data processing, including working with raw data, channel combination, B0 drift correction and water suppression.

1.2 2. Channel combination with Siemens twix data

In our previous example, we loaded data from the Siemens RDA format. While this is the default format for exporting MRS data from the scanner, Siemens also supports exporting the data in the twix format, which contains the true raw data before any processing is done to it. Working with twix data adds some complexity because we have to implement all the processing steps for ourselves, but it also gives us a lot more control over the methods we use and can really improve the final result. In this tutorial we are going to take a look at one of the first steps in the data processing workflow, channel combination.

As usual we start by importing the packages we will need, suspect of course, numpy for its general mathematical functionality, and matplotlib for visualising our results along the way. As this is a Jupyter notebook, we also use some IPython magic to render our plots interactively inside the notebook.

[1]:
```
import suspect
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

Just as in the previous tutorial we will be using data from the Suspect test data set. In this example we are using data in the twix VB format; the VD and VE formats are also supported and are generally very similar.

[2]:
```
data = suspect.io.load_twix("/home/jovyan/suspect/tests/test_data/siemens/twix_vb.dat")
```

If we inspect the shape of the loaded data we find that there are actually 3 dimensions.

[3]:
```
data.shape
```

[3]:
```
(128, 32, 2048)
```

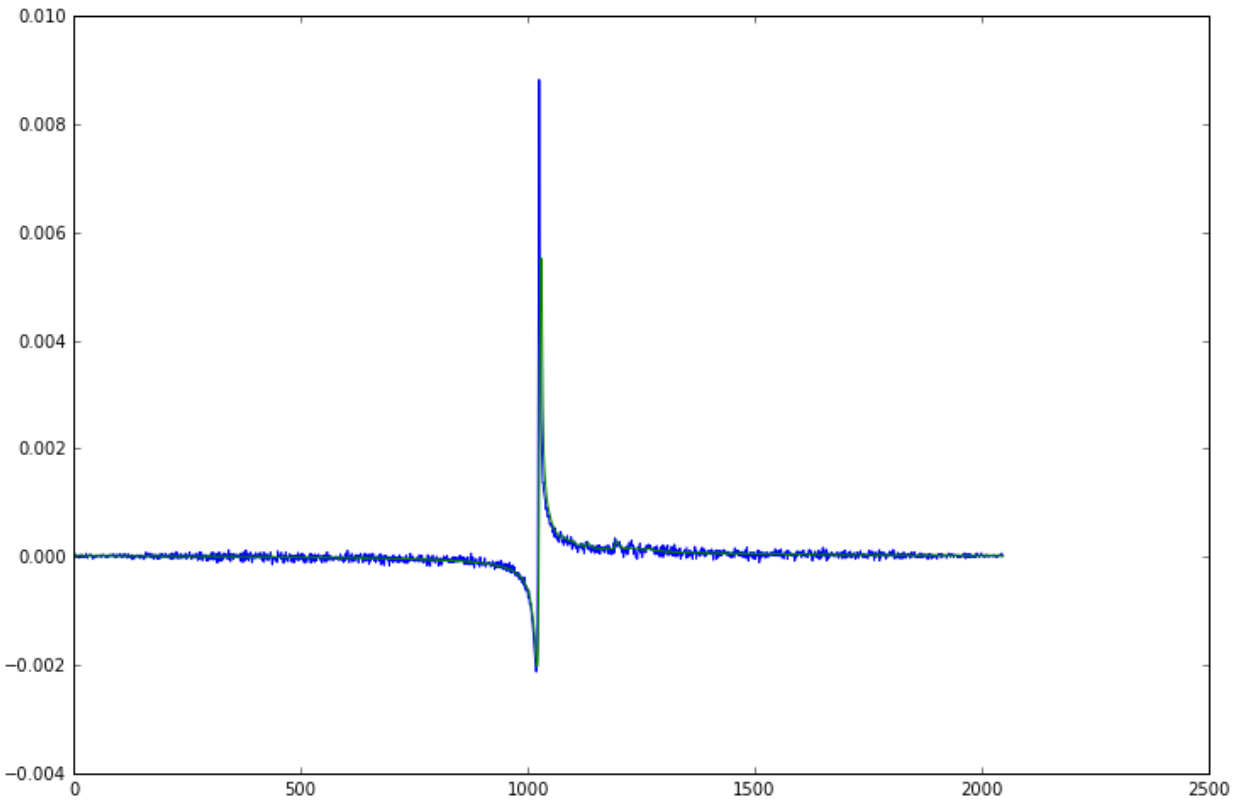
- The first dimension is the repetition index. Because of the low SNR in MRS, it is very common to perform the same acquisition multiple times and average the results. Because the signal is the same for every repetition, but since the noise varies randomly, overall the SNR improves by the square root of the number of repetitions. The downside to this is the increased acquisition time due to the extra repetitions. In this case we use 128 repetitions for an increase in SNR of ~11 times.
- The second dimension is the channel index. For this exam we used a 32 channel head coil.
- The final dimension is the points in the FID.

The first processing step we are going to carry out is to average over the repetitions. Each repetition is produced by running the exact same sequence so we can simply take the mean along this first axis to get a much less noisy signal.

[4]:
```
repetition_averaged_data = np.mean(data, axis=0)
```

[5]:
```
plt.plot(np.real(data[0, 0].spectrum()))
plt.plot(np.real(repetition_averaged_data[0].spectrum()))
```

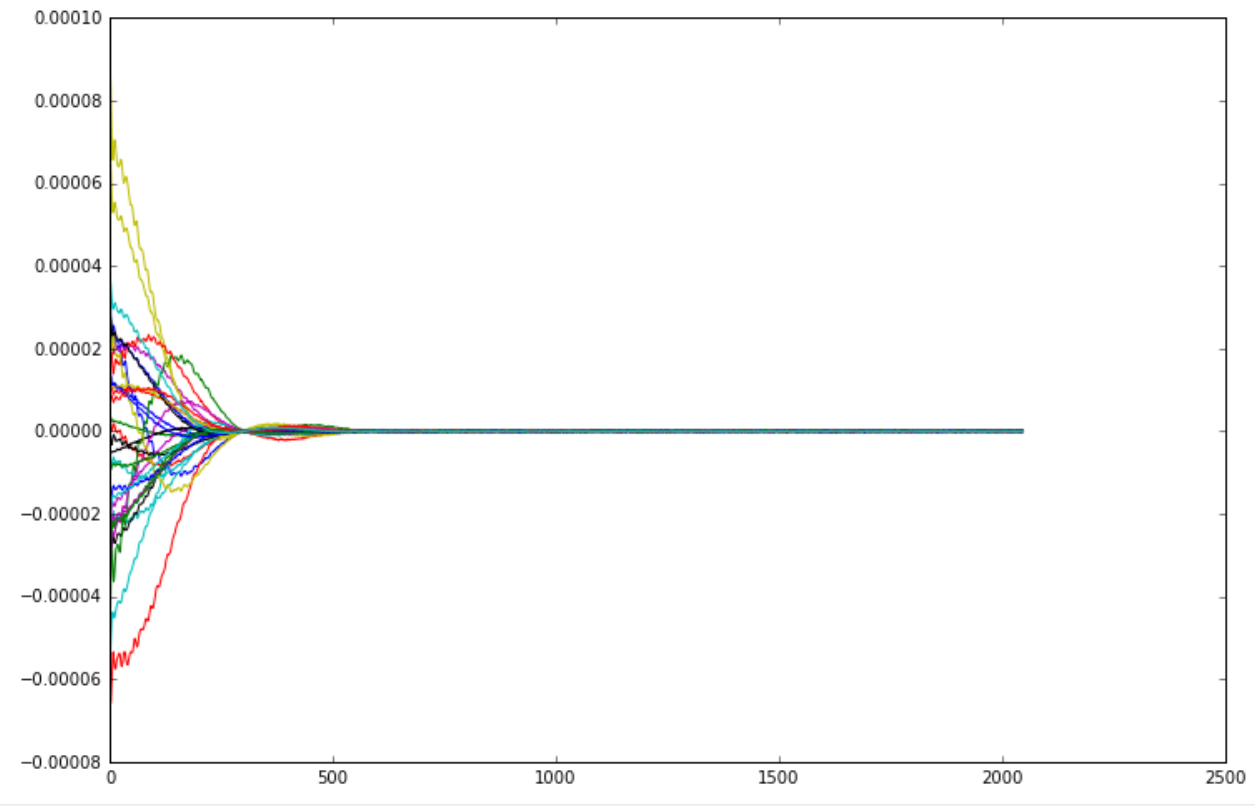
[5]:
```
[<matplotlib.lines.Line2D at 0x7f09d5b013c8>]
```



Now that we have averaged all the repetitions, we have a 2D dataset of 32 FIDs, one for each channel of our head coil. You might expect that we can just average the channels in exactly the same way, after all they are all measuring the

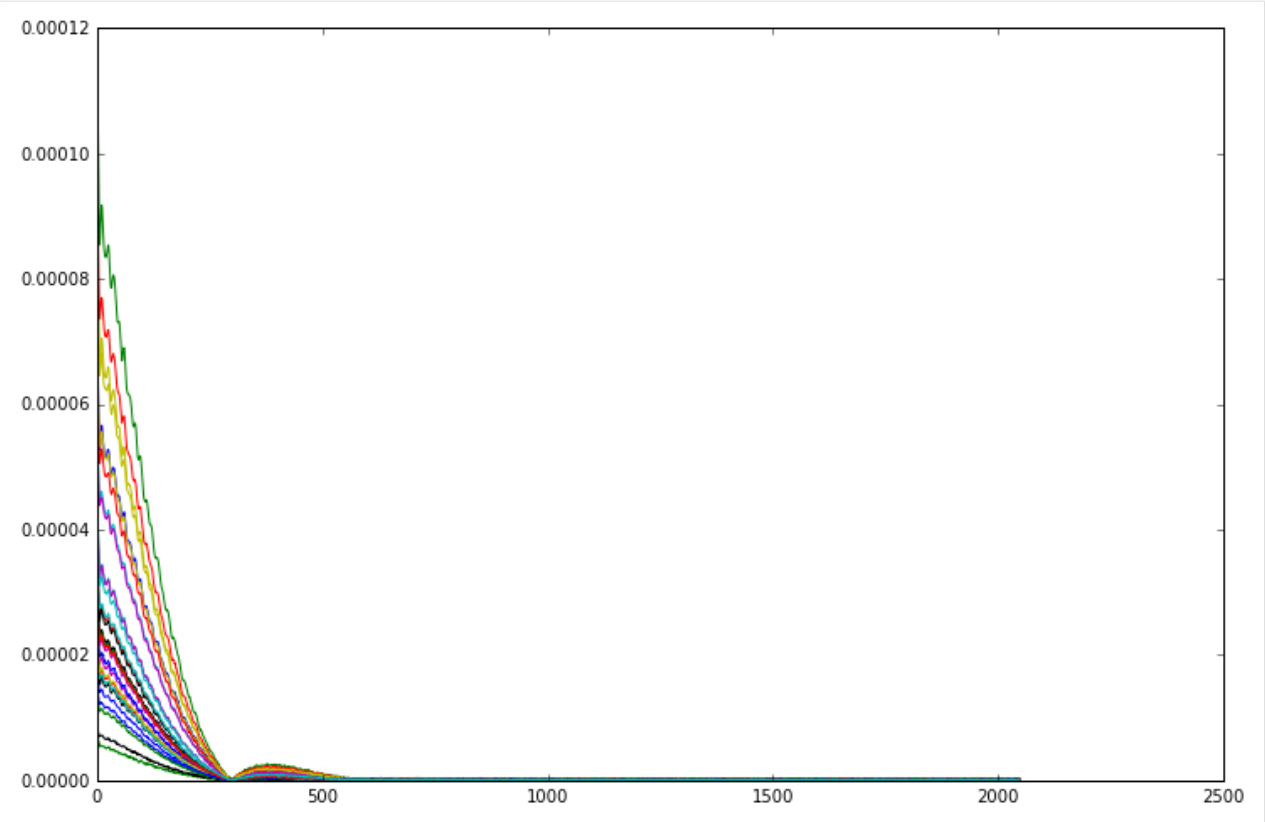
same signal at the same time. However if we plot all the FIDs together we see it isn't quite so simple as that.

[6]:
```
for i in range(32):
    plt.plot(np.real(repetition_averaged_data[i]))
```



Although all the channels are indeed measuring the same signal, each coil element is in a different position relative to the excitation voxel, and pointing in a different direction. This means that each channel measures the signal with a different sensitivity and phase. This is easier to see if we plot the magnitude of the FIDs, removing the phase component from the picture.

[7]:
```
for i in range(32):
    plt.plot(np.abs(repetition_averaged_data[i]))
```

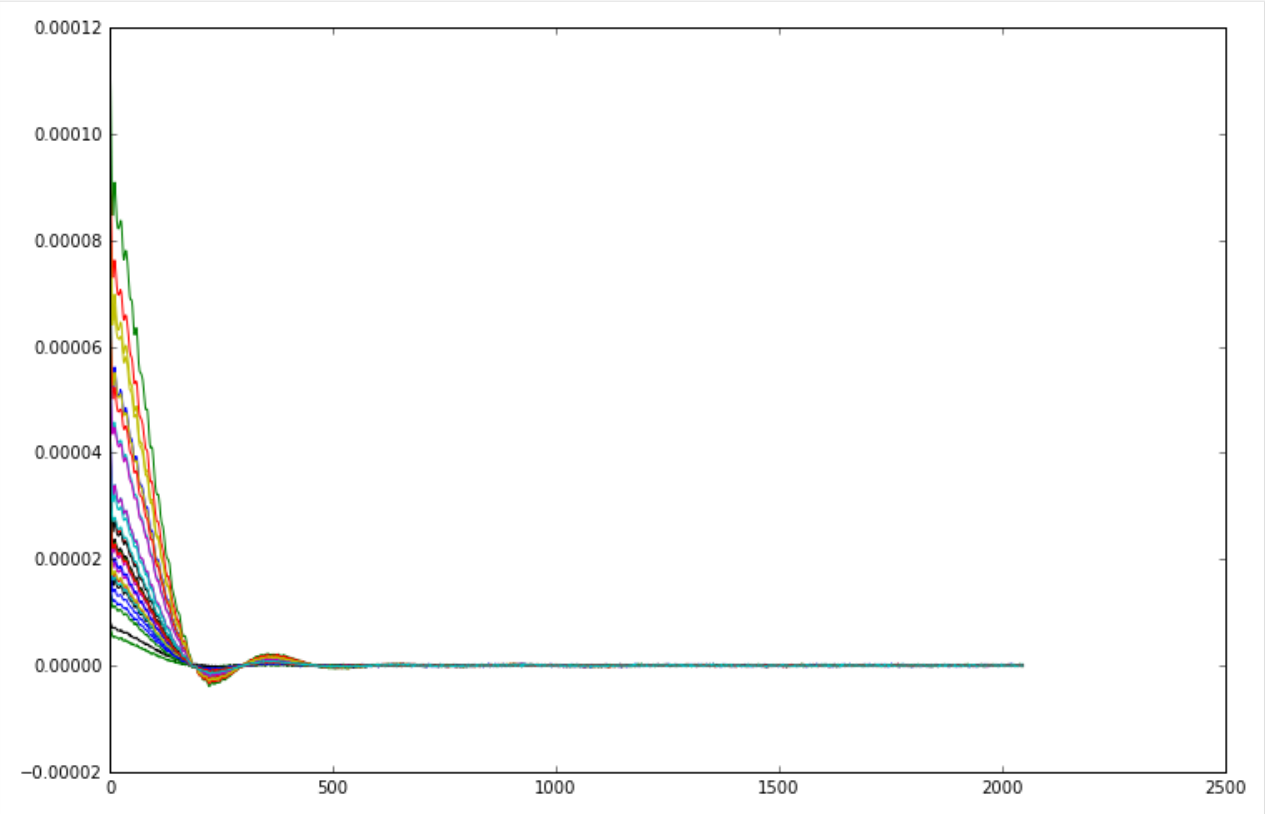


From this plot it is much easier to see that the signal from every channel has the same shape, they only differ in scale. Although in magnitude mode all the channels line up, we need to phase them before we can combine the complex FIDs.

To do this, we are going to use a very powerful technique for working with multi-dimensional MRSData objects, `numpy`'s `apply_along_axis` function. What this does is split our 2D matrix up into the individual FIDs, transform them according to a custom function, and build the results back up into a 2D matrix again. In this case, we specify a very simple function which adjusts the phase of the FID to make the first point real.

N.B. in the current version of numpy calling `apply_along_axis` will return a bare `ndarray` rather than the `MRSData` subclass, thus all the spectroscopy specific information such as f0 and dt is lost. There is a submitted pull-request to the numpy codebase to fix this issue, in the meantime we can use the `MRSData.inherit` method to copy values from the parent `MRSData` instance.

[8]:
```
def phase_correct(fid):
    return fid * np.exp(-1j * np.angle(fid[0]))
phase_corrected_data = np.apply_along_axis(phase_correct, 1, repetition_averaged_data)
# copy the MRS parameters from repetition_averaged_data to phase_corrected_data
phase_corrected_data = repetition_averaged_data.inherit(phase_corrected_data)
for i in range(32):
    plt.plot(phase_corrected_data[i].real)
```



At this point you might be tempted to apply scaling factors to each FID to make them line up in magnitude as well as in phase, however that would be a mistake. As you can see from the right hand side of the FIDs, the level of noise is the same in each one. As the size of the signal is different for every channel, that means that the signal to noise ratio is also different. Scaling up the smaller signals to match the larger will also introduce a disproportionate amount of noise into the overall signal. Instead, we should actually scale the data to make the large signals larger and the small signals smaller, as that will give us the best SNR overall. We can modify our phase correction function to also perform this scaling, then we average over the phased and scaled channels to get the final answer. If we compare the resulting spectrum against a simple unweighted average of the phase corrected FIDs, we can see a significant boost in the SNR.

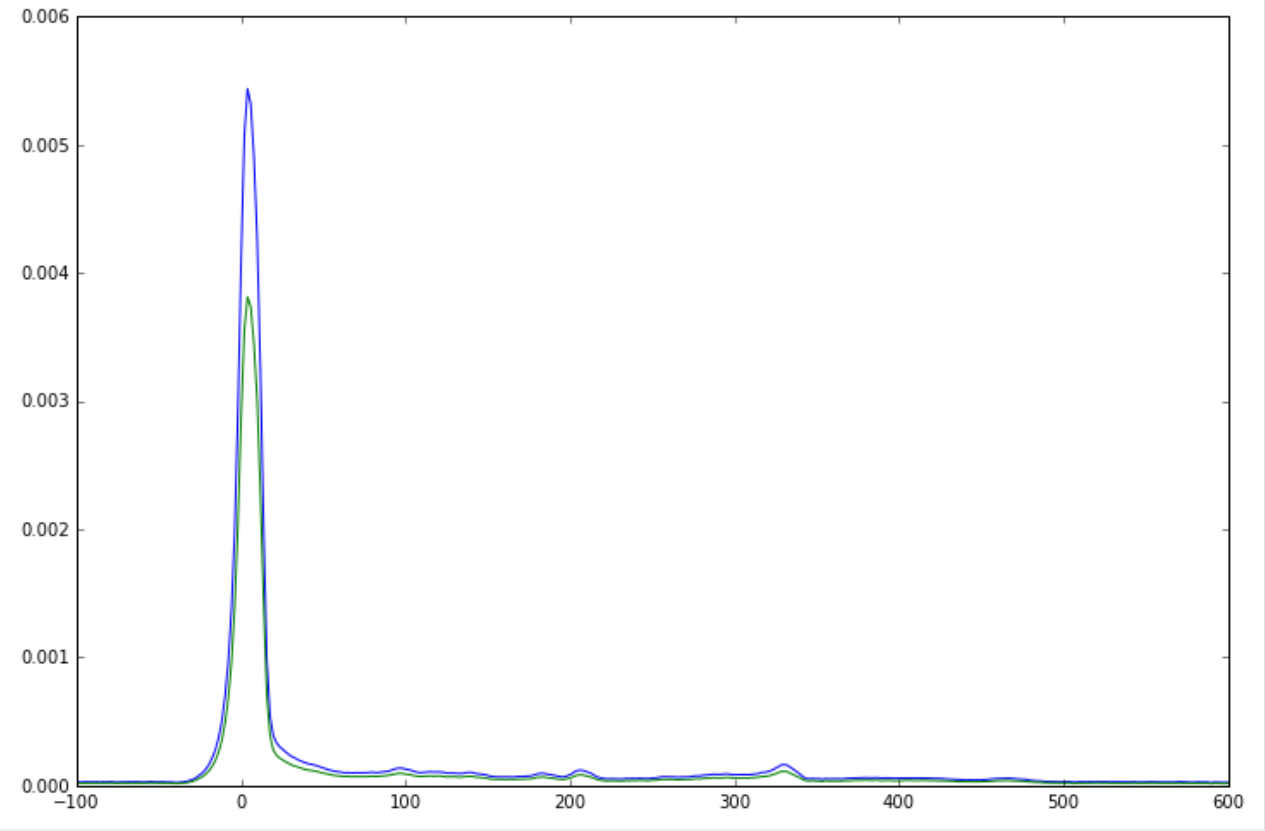
[9]:
```
def phase_correct_and_scale(fid):
    return fid * np.exp(-1j * np.angle(fid[0])) * np.abs(fid[0])
scaled_data = repetition_averaged_data.inherit(
    np.apply_along_axis(phase_correct_and_scale, 1, repetition_averaged_data)
)
# we need to normalise our scaled data by the mean scaling factor so that
# on average the noise level does not change. this allows an easier
# comparison of SNR by just looking at the two signals.
scaled_data /= np.mean(np.abs(repetition_averaged_data[:, 0]))

# average
channel_combined_data = np.mean(scaled_data, axis=0)
simple_mean = np.mean(phase_corrected_data, axis=0)

plt.plot(channel_combined_data.frequency_axis(), np.real(channel_combined_data.
↪spectrum().real))
plt.plot(simple_mean.frequency_axis(), simple_mean.spectrum().real)
# plot only the peak region to make differences easier to see
```

(continues on next page)

(continued from previous page)

```
plt.xlim([-100, 600])
```

[9]:
```
(-100, 600)
```



So far, we have have been using the first point of each channel FID to calculate both the phase and weighting of each channel. That works well for this dataset, which has a large residual water peak, but it cannot be relied upon in all cases. In data with strong water suppression, the SNR will drop significantly, while in some cases ringing from transmitted RF or gradient coils can corrupt the initial points of the FID and require them to be suppressed.

It is preferable to use a method which can estimate the optimal channel phases and weights using the complete FID as this maximises the available SNR and gives the most robust result. A good way to achieve this is with the Singular Value Decomposition (SVD). The SVD factorises a matrix M into the product of three matrices $U\Sigma V^\dagger$ where U and V are unitary and Σ is a non-negative real diagonal matrix. In the context of our 32×1024 channel $\times$ FID matrix, we can interpret $V^\dagger$ as a set of created FIDs, Σ as the amounts of each FID in the measurement, and U as the strength with which each channel measures each FID.

[10]:
```
u, s, v = np.linalg.svd(repetition_averaged_data, full_matrices=False)
```

For this kind of single voxel data we know there is only one FID being generated, all the others should just be noise. If we print out all the singular values of our decomposition, helpfully sorted into descending order by NumPy:

[11]:
```
print(s)
```

```
[  1.95606942e-03   1.68398598e-05   1.10745846e-05   8.04545743e-06
   7.00992012e-06   6.73828739e-06   6.21173379e-06   5.97860050e-06
   5.80999121e-06   5.41935388e-06   5.12515151e-06   4.92714095e-06
   4.80960584e-06   4.63161541e-06   4.50883925e-06   4.33497448e-06
   4.12607732e-06   3.98773077e-06   3.91373956e-06   3.77949787e-06
```

(continues on next page)

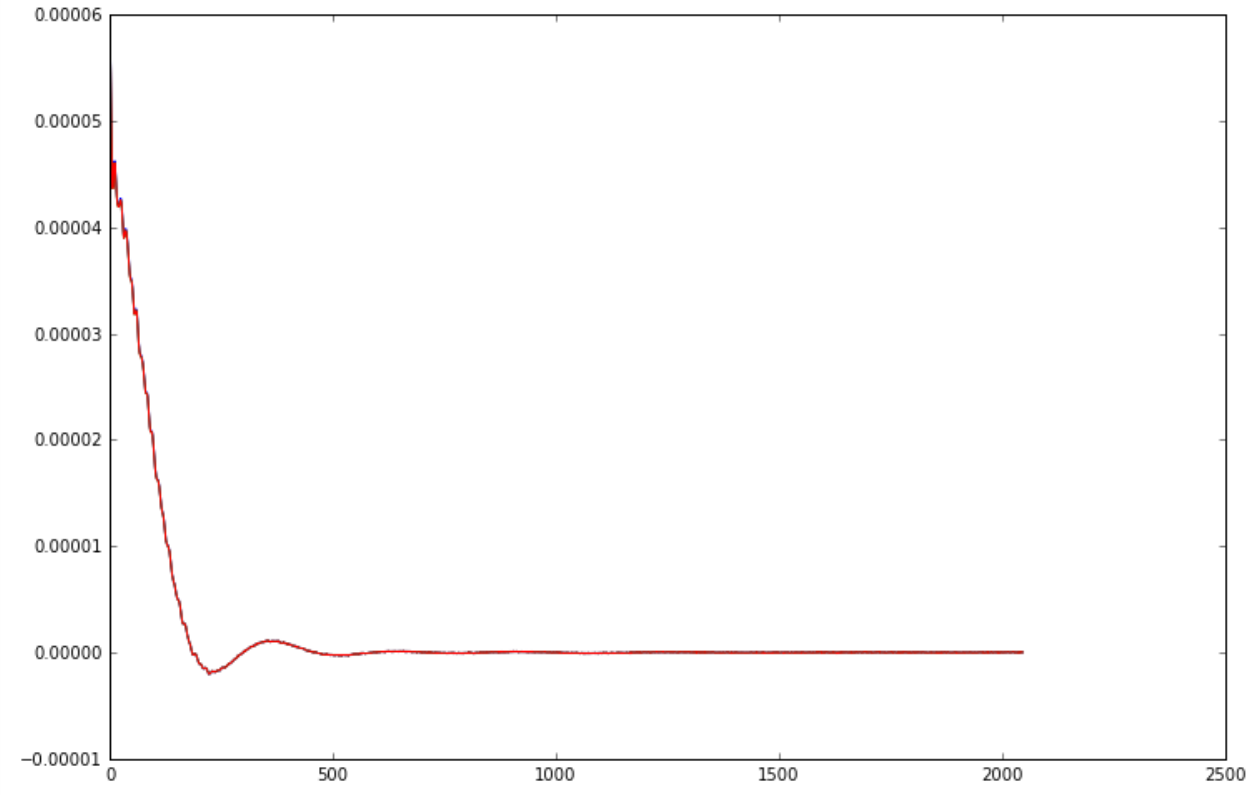
(continued from previous page)

```
   3.64188480e-06   3.62543754e-06   3.49186287e-06   3.37929421e-06
   3.24924499e-06   3.18338310e-06   3.09072752e-06   2.96630404e-06
   2.84717143e-06   2.79823675e-06   2.69785503e-06   2.57237787e-06]
```

then we see immediately that the largest singular value is more than 100 times greater than the others, this is the real FID so if we truncate u, s and v to only the first row, value and column respectively then s x the first column of v gives us the FID, and the first row of u gives us the optimal weights.

[12]:
```
weights = u[:, 0].conjugate()
# we have to normalise the weights to match the previous weight calculations
weight_normalisation = np.sum(np.abs(weights))
weights /= weight_normalisation
# the FID from the SVD has an arbitrary phase difference from the previous one
phase_shift = np.angle(v[0, 0] / channel_combined_data[0])
svd_fid = np.exp(-1j * phase_shift) * s[0] * v[0] / weight_normalisation
plt.plot(channel_combined_data.real)
plt.plot(svd_fid.real)
# we can also get the fid by multiplying the channel data by the weights
weighted_fids = weights.reshape((32, 1)) * repetition_averaged_data
# average and phase the weighted fids
svd_fid2 = np.exp(-1j * phase_shift) * np.sum(weighted_fids, axis=0)
plt.plot(svd_fid2.real)
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7f09c9f26f60>]
```

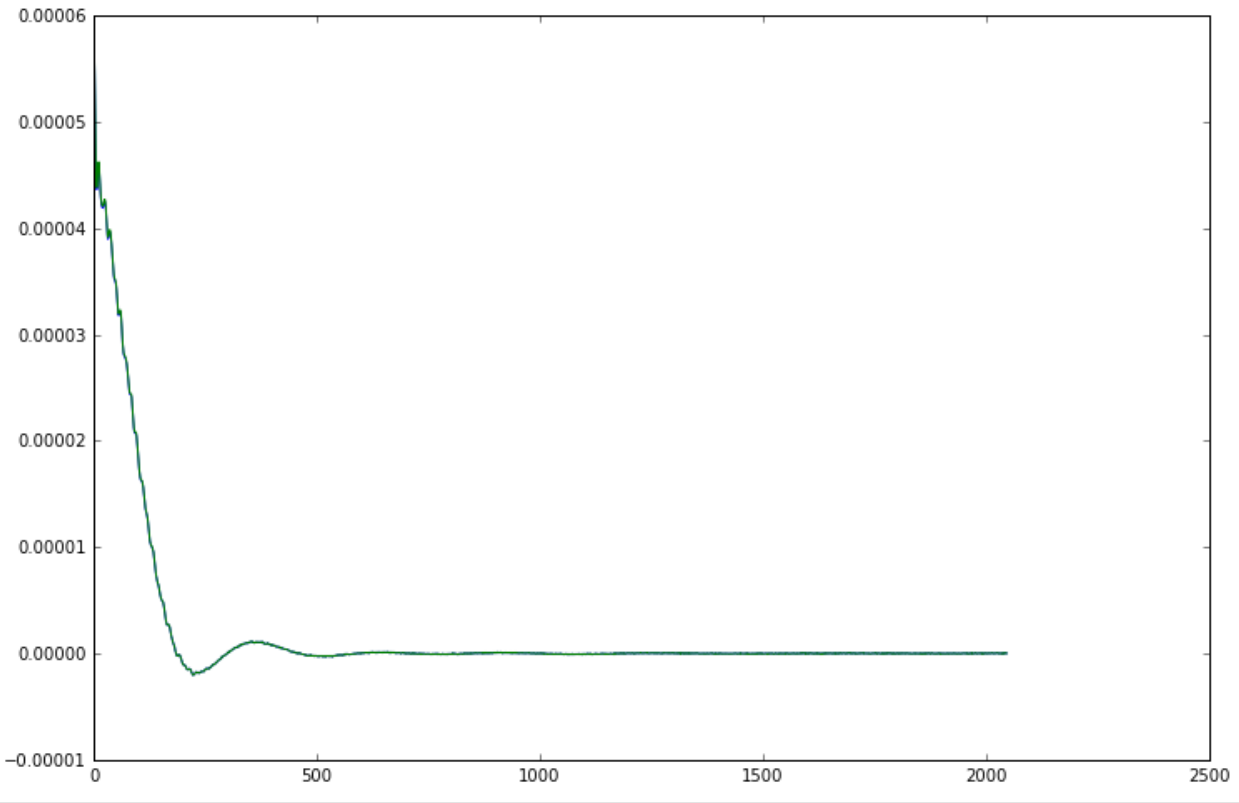


As you can see, both the FIDs we obtain from the SVD method are identical to the result we obtained previously. However, because they calculate the channel weights using the entire FID rather than just the first point, they are much more robust and perform substantially better in cases with lower SNR or corrupted initial points.

So far we have been working entirely with standard `numpy` functions, and only used Suspect to load the initial data, but of course it is also possible to use functions from Suspect's `processing.channel_combination` module to do the channel combination more simply.

[13]:
```
# first calculate the weights for each channel using the SVD method
channel_weights = suspect.processing.channel_combination.svd_weighting(repetition_
↪averaged_data)
# apply the calculated weights to the data
cc_data = suspect.processing.channel_combination.combine_channels(repetition_averaged_
↪data, channel_weights)
# the result is the same as our original, manually processed FID
plt.plot(cc_data.real)
plt.plot(channel_combined_data.real)
```

[13]:
```
[<matplotlib.lines.Line2D at 0x7f09c9f72198>]
```



In the `channel_combination` module, the channel weights can be calculated using the `svd_weighting()` function while the `combine_channels()` function applies those weights to a set of FIDs. The decision to use two functions rather than a single function which performs both steps was made to improve the flexibility of the library. By splitting the functionality it makes it easy to calculate the channel weights in a different way if necessary, or to apply one set of weights to multiple datasets. For example when an unsuppressed water reference is acquired the weights calculated using that data can also be applied to the water suppressed data, while taking advantage of the much higher SNR of the unsuppressed acquisition.

So that concludes our look at channel combination for phased array MRS data. We have learned that the different channels need to be phase corrected, and that a weighted combination of the signals can significantly improve the overall SNR. We used the SVD method to calculate both the phase corrections and the optimal weights simultaneously, and seen how Suspect provides functions to make it easy to apply. Next time, we will look at B0 shifts and how to correct for them.

1.3 4. External Quantification Tools

[1]:
```
import suspect
import numpy as np
from matplotlib import pyplot as plt
%matplotlib nbagg
```

[2]:
```
data = suspect.io.load_rda("/home/jovyan/suspect/tests/test_data/siemens/SVS_30.rda")
```

1.3.1 LCModel

LCModel requires several different files in order to process a spectrum. The actual MRS data is stored in the time domain in a .RAW file, with an optional .H20 file containing the water reference data (these are actually the same format but the extension helps in distinguishing the files). A .CONTROL file is what the LCModel program actually receives as input, it lists the input and output files, and any parameters which have non-default values. Finally the metabolite basis set used for fitting the data is contained in a .BASIS file. The .BASIS file is normally provided with LCModel and unlike the other files does not need to be changed with each new dataset.

Suspect can generate all the files necessary to process data with LCModel using the `write_all_files()` function from the `suspect.io.lcmodel` module. This function takes a path to which to save the .RAW file (any other files will be saved in the same directory), the `MRSData` object to be written, and an optional `params` dictionary to customise any of the values in the .CONTROL file. This dictionary can contain any of the parameter names from the LCModel manual. Let's take a look at an example of how to save our data for LCModel to process.

[3]:
```
# create a parameters dictionary to set the basis set to use
params = {
    "FILBAS": "/path/to/lcmodel/basis.BASIS"
}
suspect.io.lcmodel.write_all_files("lcmodel_data/example.RAW", data, params=params)
```

We can use some IPython magic to show the files that were created:

[4]:
```
!ls lcmodel_data/
```

```
example.RAW  example_sl0.CONTROL
```

and to look at the contents of the .CONTROL file:

[5]:
```
!cat lcmodel_data/example_sl0.CONTROL
```

```
 $LCMODL
 OWNER = ''
 KEY = 123456789
 DELTAT = 0.000833
 HZPPPM = 123.234655
 NUNFIL = 1024
 NDROWS = 1
 NDCOLS = 1
 FILBAS = '/path/to/lcmodel/basis.BASIS'
 NDSLIC = 1
 FILPS = 'lcmodel_data/example.PS'
 ICOLST = 1
 FILRAW = 'lcmodel_data/example.RAW'
```

(continues on next page)

(continued from previous page)

```
 IROWEN = 1
 ICOLEN = 1
 DOWS = F
 IROWST = 1
 DOECC = F
 $END
```

The .CONTROL file contains all the parameters necessary for LCModel to process the file, including the path we specified to the correct basis set. Once the .RAW and .CONTROL files are generated, it only remains to run the LCModel program on the command line, passing in the .CONTROL file like this:

```
lcmodel < lcmodel_data/example_sl0.CONTROL
```

If you are running your Suspect code on the same computer as your LCModel installation, so that lcmodel is in your path, it is trivial to run it from within a Jupyter notebook using IPython magic, or from a script using the `subprocess` module. However if you are running Suspect on a different computer, for example using the OpenMRSLab Docker container, then you will have to transfer the LCModel files to the LCModel computer for processing. This can be inconvenient because the paths in the generated .CONTROL file can become out of date. In our lab we have solved this problem by using a shared network drive which is mounted by both machines, then the paths remain consistent and the `lcmodel` program can be launched remotely from the Docker container with a simple ssh command. Another alternative is to use the `suspect.io.lcmodel.save_raw()` function instead to save only the .RAW file which can then be loaded by LCMGUI and the other parameters configured there.

By default the control file is set to only generate the standard 1 page .PS output. The other output files can be generated by setting the appropriate options in the params dictionary. For example, to generate the .CSV file, set the "LCSV" parameter to True. In this case, Suspect will automatically generate the path for the .CSV in the same folder as the .RAW and .CONTROL files. To set a custom location to save the .CSV, instead set the parameter "FILCSV" in params.

[6]:
```
# create a parameters dictionary to set the basis set to use
params = {
    "FILBAS": "/path/to/lcmodel/basis.BASIS",
    "LCSV": True
}
suspect.io.lcmodel.write_all_files("lcmodel_data/example.RAW", data, params=params)
```

[7]:
```
!cat lcmodel_data/example_sl0.CONTROL
```

```
 $LCMODL
 OWNER = ''
 KEY = 123456789
 DELTAT = 0.000833
 HZPPPM = 123.234655
 NUNFIL = 1024
 NDROWS = 1
 NDCOLS = 1
 FILBAS = '/path/to/lcmodel/basis.BASIS'
 NDSLIC = 1
 FILPS = 'lcmodel_data/example.PS'
 ICOLST = 1
 FILRAW = 'lcmodel_data/example.RAW'
 IROWEN = 1
 ICOLEN = 1
 LCSV = 11
```

(continues on next page)

(continued from previous page)

```
DOWS = F
IROWST = 1
FILCSV = 'lcmodel_data/example.CSV'
DOECC = F
$END
```

1.4 5. Water suppression with HSVD

In this tutorial we will take a look at water suppression. Water is present in the body at concentrations thousands of times higher than any of the metabolites we are interested in, so any spectrum where the water signal is not suppressed is completely dominated by the water peak centred at 4.7ppm.

The standard way to suppress the water is to use the CHESS (chemical shift selective) technique. This preparation method uses a frequency selective excitation pulse to excite only the spins in the region of the water peak, followed by a "crusher" gradient pulse which dephases the excited spins. Once they have lost their phase coherence, these spins will no longer contribute any signal during the acquisition. In practice, the basic CHESS technique has been superseded by first WET and now VAPOR, which use a sequence of CHESS style pulses with varying flip angles and delays to achieve greater tolerance to B1 variation, and generally improved performance.

However, in many cases, this prospective water suppression is insufficient to completely remove the water signal. Regions with poor shim, such as tumour, may have a water peak which partially extends outside the suppression region, and patient movement can have the same effect. Furthermore, many users choose to reduce the effect of water suppression by allowing a small amount of T1 recovery between the CHESS and the acquisition sequence. This approach, often referred to as "weak" water suppression, gives a large residual water peak which is useful during processing, for calculating channel weights and correcting frequency shifts. This peak must then be removed in a further processing step.

The methods available for removing the residual water peak generally involve some form of bandpass filter which removes the signal from a particular region of the spectrum. For this tutorial we are going to focus on the most widely used technique, HSVD (Hankel Singular Value Decomposition).

As usual, we start by importing our dependencies:

[1]:
```
import suspect
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

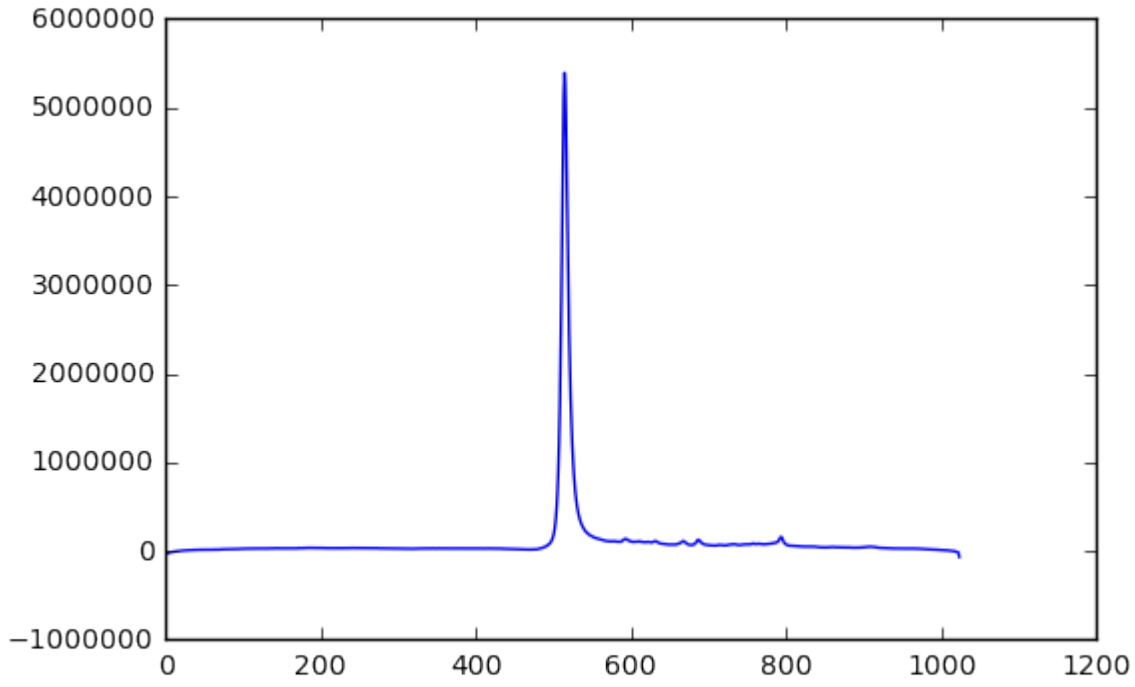
For this tutorial, we will be using the SVS_30.rda data included in the Suspect test data collection, so that we don't have to worry about channel combination or frequency correction here. However, we will repeat the apodisation step described in Tutorial 1.

[2]:
```
data = suspect.io.load_rda("/home/jovyan/suspect/tests/test_data/siemens/SVS_30.rda")

import scipy.signal
window = scipy.signal.tukey(data.np * 2)[data.np:]
data = window * data
```

If we plot the raw spectrum we immediately see that the water peak completely dominates all the other peaks in the spectrum:

[3]:
```
plt.plot(data.spectrum().real)
```

[3]:
```
[<matplotlib.lines.Line2D at 0x7fbeb58a1c88>]
```



HSVD works by approximating the FID with a set of exponentially decaying components:

[4]:
```
components = suspect.processing.water_suppression.hsvd(data, 20)
```

The second argument to the function is the number of components to generate. This will depend on both the number of peaks in the spectrum and how Lorentzian they are. Too few components will not be able to correctly describe the signal but too many can lead to over-fitting. Around 20 is typically a good number for most cases, but do experiment with your own data to understand better exactly what is going on.

The `hsvd()` function returns a `list` of `dict`s, with each `dict` containing information about one exponential component:

[5]:
```
print(components[0])
```

```
{'phase': -2.3526332899894427, 'amplitude': 137.25589213068457, 'fwhm': 0.
↪93358884978978407, 'frequency': 598.57586504857886}
```

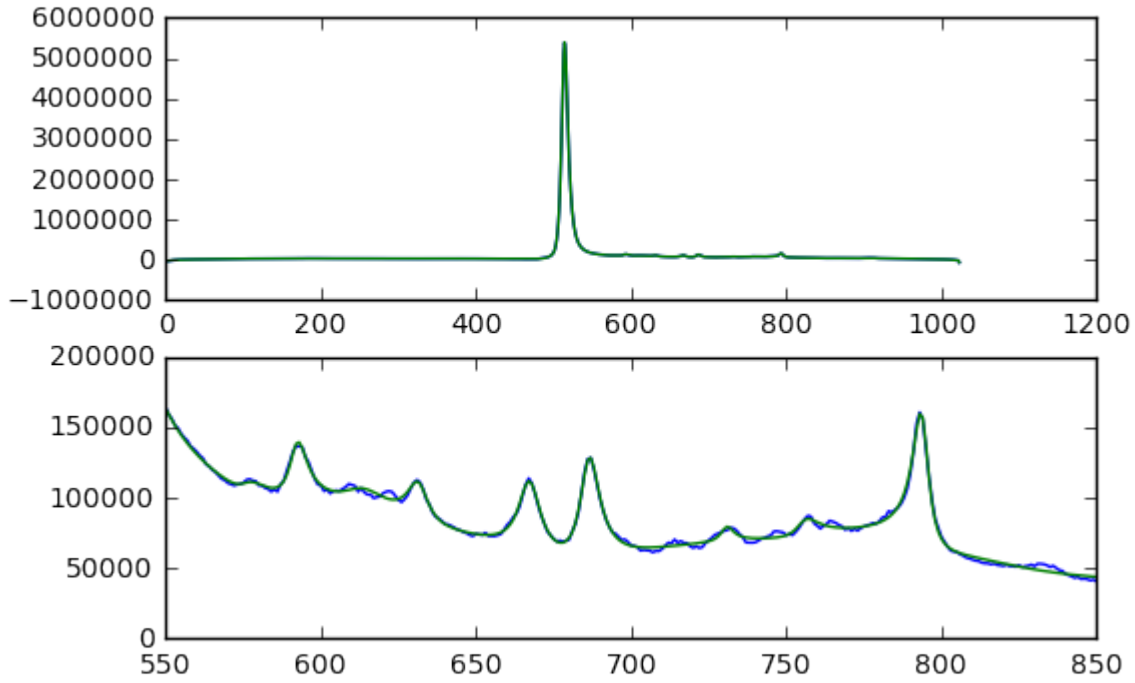
This components `list` can be turned back into an FID using the `construct_fid()` function, which takes a list of components to be used and a reference time axis. In this example we also set the resulting FID to `inherit()` all the MRS properties from the original `data` object.

[6]:
```
hsvd_fid = suspect.processing.water_suppression.construct_fid(components, data.time_
↪axis())
hsvd_fid = data.inherit(hsvd_fid)
# plot two axes, one of the whole spectrum and one focussing on the metabolite region
f, (ax1, ax2) = plt.subplots(2)
ax2.set_xlim([550, 850])
ax2.set_ylim([0, 2e5])
for ax in (ax1, ax2):
```

(continues on next page)

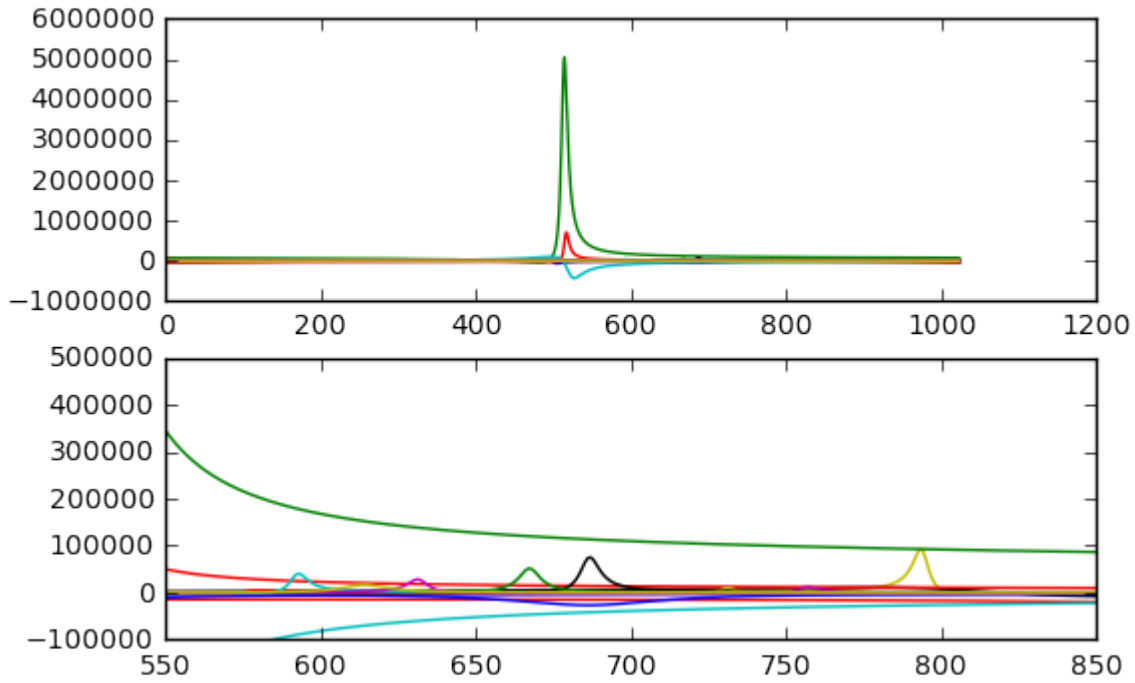
(continued from previous page)

```
    ax.plot(data.spectrum().real)
    ax.plot(hsvd_fid.spectrum().real)
```



Overall we see that the `hsvd_fid` is a very good approximation to the original `data` signal, although some of the smaller peaks such as the Glx region are not fitted. To get a better idea of what is going on, we can reconstruct each component individually and plot the whole set together.

[7]:
```
# plot two axes, one of the whole dataset and one of the metabolite region
f, (ax1, ax2) = plt.subplots(2)
ax2.set_xlim([550, 850])
ax2.set_ylim([-1e5, 5e5])
for component in components:
    component_fid = suspect.processing.water_suppression.construct_fid([component], data.
↪time_axis())
    component_fid = data.inherit(component_fid)
    ax1.plot(component_fid.spectrum().real)
    ax2.plot(component_fid.spectrum().real)
```



What we find is that the major metabolite peaks each have one component associated with them, while the water peak has several. This is because it is not a perfect Lorentzian - to adequately describe the peak shape requires a series of progressively smaller correction terms to modify the main peak. Typically only the water peak gets multiple components as the others are too small, and the total number of components is limited.

The next step is to separate out the components making up the water signal from the metabolite components, which we do using a frequency cut-off. We can do this rather neatly using a Python list comprehension:

[8]:
```
water_components = [component for component in components if component["frequency"] < 70␣
↪or component["fwhm"] > 100]
```

In this case we have selected all the components with frequencies below 80Hz. The best value for this cut-off frequency will depend strongly on your data, and of course on the field strength of the magnet, but 80Hz is a reasonable starting point for most people at 3T. For our data we don't have any peaks downfield of water so we don't need a negative frequency cut-off.

In addition we have selected a second set of components, this time with a FWHM greater than 100Hz. These very broad components are part of the baseline and it can be helpful to remove these at the same time.

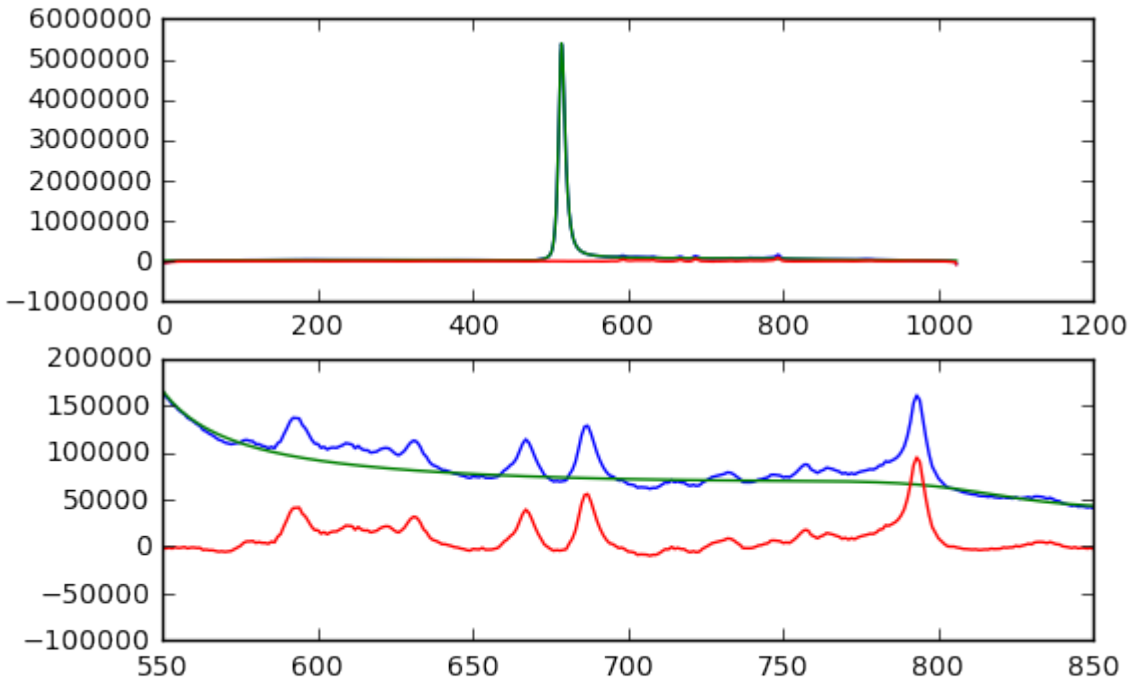
Once we have selected the components we want to remove, we can simply subtract the constructed FIDs from our original data to arrive at the water suppressed spectrum.

[9]:
```
water_fid = suspect.processing.water_suppression.construct_fid(water_components, data.
↪time_axis())
water_fid = data.inherit(water_fid)
dry_fid = data - water_fid
# plot two axes, one of the whole spectrum and one focussing on the metabolite region
f, (ax1, ax2) = plt.subplots(2)
ax2.set_xlim([550, 850])
ax2.set_ylim([-1e5, 2e5])
for ax in (ax1, ax2):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(data.spectrum().real)
    ax.plot(water_fid.spectrum().real)
    ax.plot(dry_fid.spectrum().real)
```



The `water_fid` accurately matches the residual water peak and the broad baseline, but does not include any of the metabolite peaks. Subtracting this from our data leaves a "dry" spectrum, without any remaining water contamination, ready for the next stage in the processing pipeline.

1.5 6. Image co-registration

One of the most important steps in MRS processing is visualising the spectroscopy region on a structural image. This not only allows us to validate that the voxel was correctly placed and assess any partial volume effects, but can also be used to estimate the mean T2 of the voxel from the gray and white matter content (for brain spectroscopy). Quantification tools use the T2 to correct for signal relaxation and adjust the metabolite concentrations, but by default they use a generic value. By calculating your own from the voxel location, you can get more accurate concentrations. Co-registration is also particularly important for spectroscopic imaging, for example when generating heatmap overlays on top of structural images.

In this tutorial, we will look at how to relate the coordinate systems of the scanner, the image and the spectroscopy together, and how to plot a simple voxel outline on top of an image slice. In later tutorials, we will see how to use this for T2 estimation, and look at more advanced renderings for CSI data.

[1]:
```
import suspect
import numpy as np
import matplotlib.pyplot as plt
```

Loading structural MR images is handled by the `suspect.image` module. Currently, suspect only supports working with DICOM files, although support for NIFTI and other formats is planned for the future. The `load_dicom_volume()`

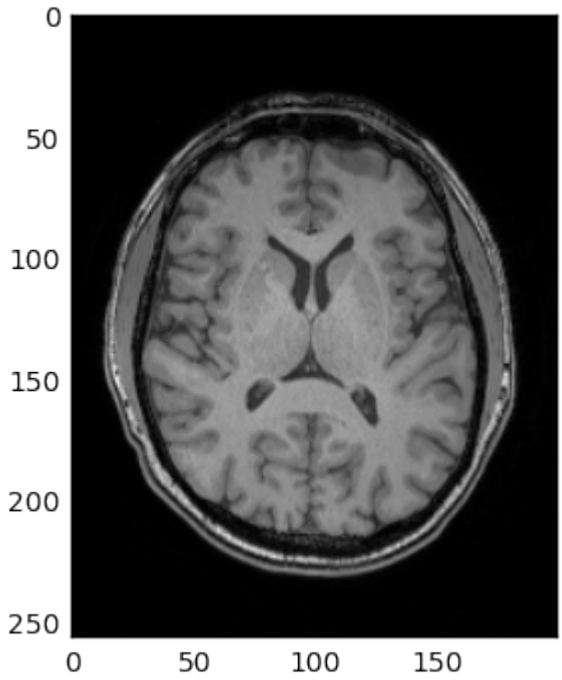
function finds all the DICOM files in the directory of the specified file which have the same SeriesUID, and loads them all into a single 3D volume of the `ImageBase` class.

[2]:
```
t1 = suspect.image.load_dicom_volume("mprage/IM-0001-0001.IMA")
```

Like the MRSData class, the ImageBase class is a numpy.ndarray subclass with a few extra helper functions. This makes it very easy to display image slices using matplotlib's `imshow()` function.

[3]:
```
plt.imshow(t1[100], cmap=plt.cm.gray)
```

[3]:
```
<matplotlib.image.AxesImage at 0x7fb1c3d10390>
```



For this tutorial, the two most important functions we are going to look at are `to_scanner()` and `from_scanner()`. These functions are used to convert between the voxel coordinate system of the image, and the intrinsic coordinates of the scanner. For example, we can take the bottom left hand voxel of the image: (0, 0, 0) and see where that is located inside the scanner.

[4]:
```
origin_position = t1.to_scanner(0, 0, 0)
print(origin_position)
```

```
[ -99.75786927 -130.17917679 -103.61138058]
```

`origin_position` gives us the location of the (0, 0, 0) voxel relative to the scanner isocenter, in mm, using the standard scanner coordinates, with x increasing from right to left, y increasing from anterior to posterior and z increasing from inferior to superior.

On the other hand, we can also work out which voxel is closest to isocenter using the `from_scanner()` function.

[5]:
```
isocenter_voxel = t1.from_scanner(0, 0, 0)
print(isocenter_voxel)
```

```
[  99.75786924  130.17917681  103.61138058]
```

Note that we don't get integer values for the voxel coordinates, this is important when we want to chain coordinate

systems together later on. However, if we want to slice into our volume we can convert them to integers (making sure to round first so that e.g. 99.75 goes to 100 and not 99):

[6]:
```
isocenter_indices = isocenter_voxel.round().astype(int)
print(isocenter_indices)
```

```
[100 130 104]
```

A quick word here about `ImageBase` coordinates. Conceptually suspect loads the DICOM volume as a stack of slices which means that voxels are indexed as `volume[slice, row, column]`. This is done to make it easier to plot the images with matplotlib where the outer dimension is the vertical. However, the `to_scanner()` and `from_scanner()` functions both stick to the standard order of x, y, z, or column, row, slice. Therefore when accessing a particular slice of an image, the coordinates must be reversed. This is confusing but better than any of the alternatives so far proposed.

Now that we have seen how to load an image and understand its coordinate systems, it is time to look at the spectroscopy side. Fortunately, `MRSData` is a subclass of `ImageBase`, so the spectroscopy object has exactly the same `to_scanner()` and `from_scanner()` methods available. In this example, we will be using a voxel acquired from the posterior cingulate gyrus, in the rda format.

[7]:
```
pcg = suspect.io.load_rda("pcg.rda")

pcg_centre = pcg.to_scanner(0, 0, 0)
print(pcg_centre)
```

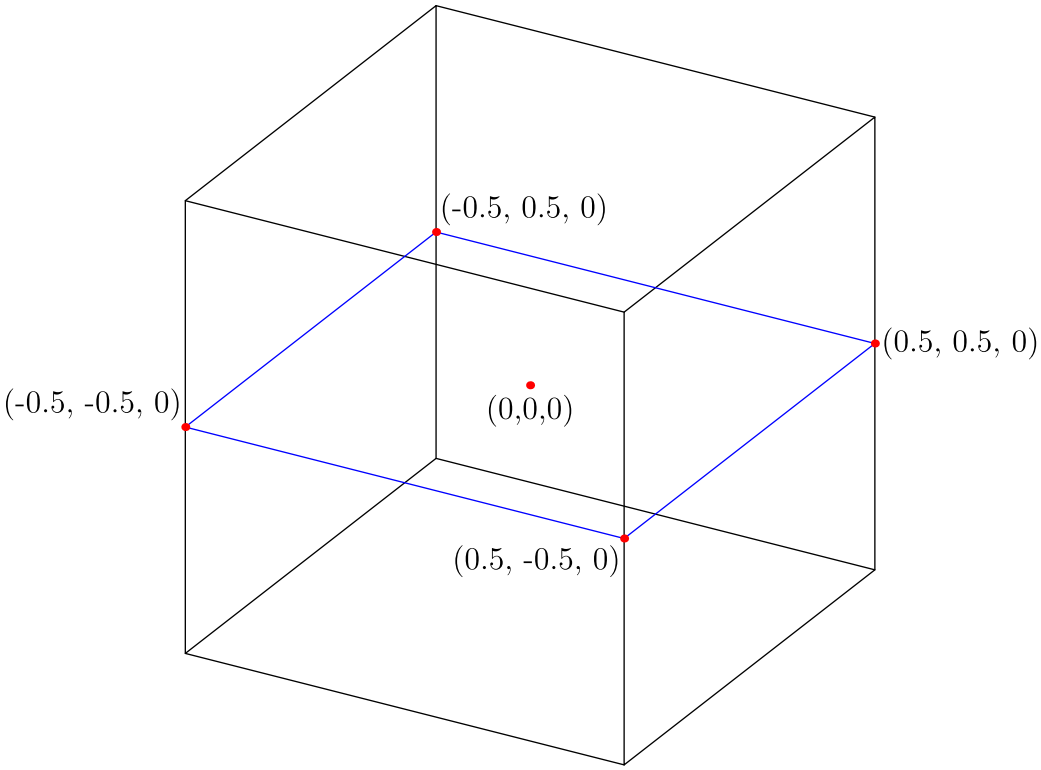
```
[ -2.542373  43.220339  21.186441]
```

As we can see, this voxel is located slightly behind and above the isocenter position. Now that we know the location of the centre of the voxel in scanner coordinates, we can convert that into the image space to tell us which slice goes through the middle of the voxel.

[8]:
```
pcg_centre_index = t1.from_scanner(*pcg_centre).round().astype(int)
print(pcg_centre_index)
```

```
[ 97 173 125]
```

This tells us that slice 125 goes through the centre of the voxel. In this case the voxel is aligned with the plane of the image, so we don't have to worry about drawing all the sides. Instead we will draw the transverse slice through the middle of the voxel, as shown in this diagram.



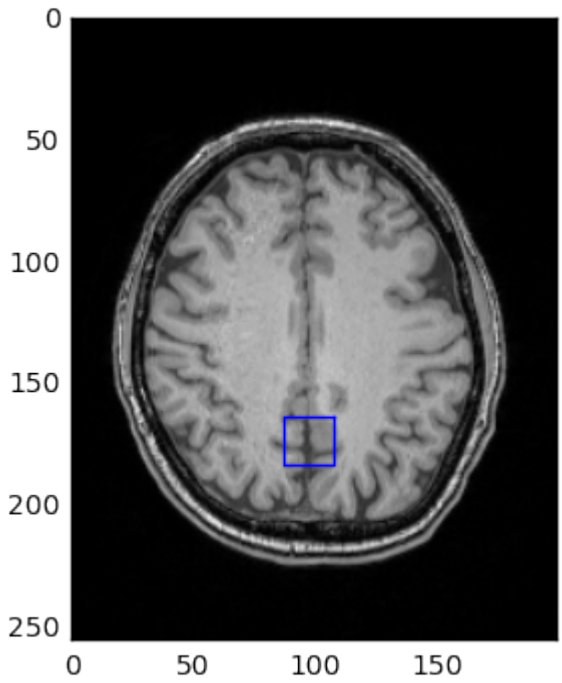
[9]:
```
corner_coords_pcg = [[-0.5, -0.5, 0],
                     [0.5, -0.5, 0],
                     [0.5, 0.5, 0],
                     [-0.5, 0.5, 0],
                     [-0.5, -0.5, 0]]
corner_coords = np.array([t1.from_scanner(*pcg.to_scanner(*coord)) for coord in corner_
↪coords_pcg])
```

We start with the list of points in the spectroscopy voxel coordinates. Note that we have added a second copy of the first point at the end to make it easier to plot a loop. We then use a list comprehension to convert each point first into the scanner coordinates and then into the image space. Finally we convert the list of coordinates into a numpy array to make it easier to access the data from matplotlib.

Now we are ready to plot the voxel. We start by displaying the correct image slice with `imshow()`, then pass the x and y coordinates of the voxel to `plot()`. As a final touch, we use the `xlim()` and `ylim()` functions to restrict the axes only to the image range, otherwise matplotlib will pad the image with blank space.

[10]:
```
plt.imshow(t1[pcg_centre_index[2]], cmap=plt.cm.gray)
plt.plot(corner_coords[:, 0], corner_coords[:, 1])
plt.xlim([0, t1.shape[2] - 1])
plt.ylim([t1.shape[1] - 1, 0])
```

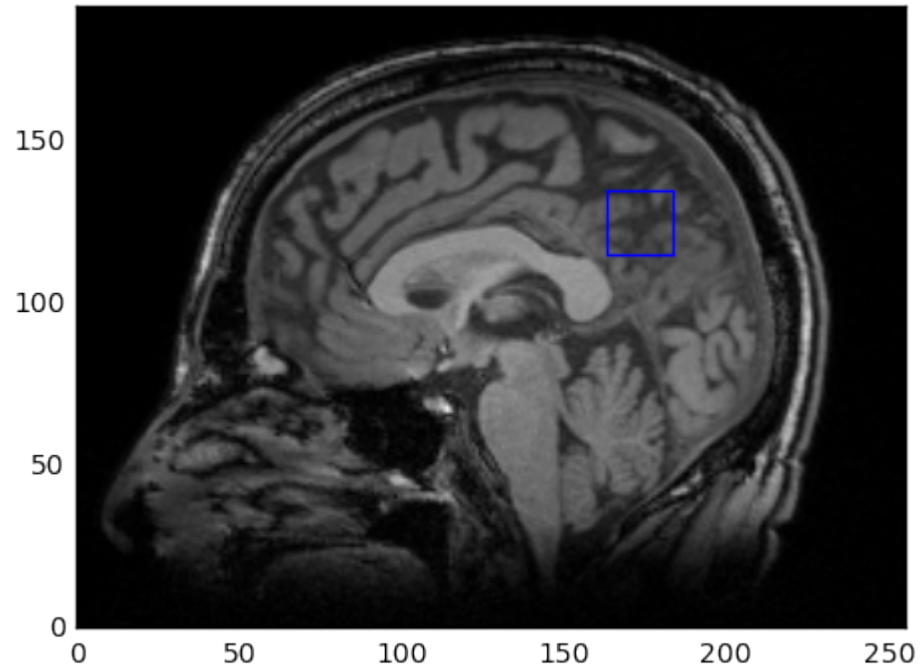
[10]:
```
(255, 0)
```



And that is really all there is to it. Now that we know how to convert spectroscopy points into image space, it is very easy to get the voxel on coronal or sagittal images as well. We just have to change the points on the spectroscopy voxel and take a different slice through the image volume. The one important thing to remember is that for sagittal and coronal images the vertical axis is z, which increases from inferior to superior, unlike the default in matplotlib where images are plotted downwards. This means we have to use `ylim()` to reverse the direction of plotting, to get our images the right way up.

[11]:
```
sagittal_voxel = [[0, -0.5, -0.5], [0, 0.5, -0.5], [0, 0.5, 0.5], [0, -0.5, 0.5], [0, -0.
↪5, -0.5]]
sagittal_positions = np.array([t1.from_scanner(*pcg.to_scanner(*coord)) for coord in␣
↪sagittal_voxel])
plt.imshow(t1[:, :, pcg_centre_index[0]], cmap=plt.cm.gray)
plt.plot(sagittal_positions[:, 1], sagittal_positions[:, 2])
plt.xlim([0, t1.shape[1] - 1])
plt.ylim([0, t1.shape[0] - 1])
```

[11]:
```
(0, 191)
```


150
100
50
0
0
50
100
150
200
250

CHAPTER

TWO

SOLVING SPECIFIC PROBLEMS

2.1 Co-registering Images

In this guide, we explore how to combine your MRS data with companion anatomical MR images. We create a voxel mask to identify which voxels of an image fall inside the spectroscopy voxel, and how to visualise the voxel on top of the image, both as a volume and an outline. We also look at resampling the image to better align with the spectroscopy voxel.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import suspect
```

Here we load the data, in this case a DICOM image set and a GE P-file. For the DICOM volume we pass a single file and the containing folder is processed to collect any other files from the same series, these are then collected into a 3D volume in a `suspect.ImageBase` class, which provides access to a lot of useful functionality around transforms and co-registering. In this case the exam is a single voxel PRESS in the hippocampus, which is convenient as it is a double oblique and very oblong voxel.

[2]:
```
image = suspect.image.load_dicom_volume("00004_MPRAGE/i5881167.MRDC.15.img")
data, wref = suspect.io.load_pfile("P75264.e02941.s00007.7")
```

We start by constructing a voxel mask, passing in the spectroscopy data which defines the excitation region and an image volume which defines the coordinate system for the mask. For display purposes in Matplotlib we then create a "masked" mask using the Numpy MaskedArray class, so that the mask will only be displayed where the mask is true.

[3]:
```
voxel_mask = suspect.image.create_mask(data, image)
masked_mask = np.ma.masked_where(voxel_mask == False, voxel_mask)
```

To draw the voxel outline we must define a set of vertices defining a path which will trace all the edges of the voxel. The list below is the most efficient ordering I can come up with, but feel free to raise an issue on GitHub if you can think of a better one. I am planning to bring this inside Suspect as a property of the MRSBase class so that it is readily available. Note that because this is in the MRS coordinates, the corners of the voxel are universal, irrespective of the size and shape of the voxel.

[4]:
```
corner_coords_voxel = [[-0.5, -0.5, -0.5],
                       [0.5, -0.5, -0.5],
                       [0.5, 0.5, -0.5],
                       [-0.5, 0.5, -0.5],
                       [-0.5, -0.5, -0.5],
                       [-0.5, -0.5, 0.5],
                       [0.5, -0.5, 0.5],
```

(continues on next page)

(continued from previous page)

```
                       [0.5, -0.5, -0.5],
                       [0.5, -0.5, 0.5],
                       [0.5, 0.5, 0.5],
                       [0.5, 0.5, -0.5],
                       [0.5, 0.5, 0.5],
                       [-0.5, 0.5, 0.5],
                       [-0.5, 0.5, -0.5],
                       [-0.5, 0.5, 0.5],
                       [-0.5, -0.5, 0.5]]
```

Here we transform the coordinate system into the coordinates of the image volume. First we use the `to_scanner()` method of the MRS `data` to move the locations of the voxel corners into the scanner space, then use `image.from_scanner()` to bring them into the coordinates of the image volume, which are the voxel indices.

[5]:
```
corner_coords = np.array([image.from_scanner(*data.to_scanner(*coord)) for coord in␣
↪corner_coords_voxel])
```

The final bit of preparation we are doing here is to calculate where the centre of the MRS voxel is located in the image volume so we know which slice (in each plane) we need to render at the centre of the voxel. Note that coordinates are a bit tricky here - data is organised on the scanner with slices as the outer loop, then columns and then rows, conceptually that means that the z coordinate is the first index into the array, then y and then x. However all manipulation of coordinates in transforms assumes the more conventional ordering of (x, y, z). Thus we find that the centre of the voxel is at slice 58, for example.

[6]:
```
voxel_centre_in_image = image.from_scanner(data.position).round().astype(int)
print(voxel_centre_in_image)
```

```
[124 132  58]
```

Now we are ready to actually render the voxels overlaid with the masks and/or voxel outlines. The anatomical scan we started with was a sagittal volume, but of course we can easily extract orthogonal slices either axially or coronally, and in this case the scan is isotropic so we don't even have to worry about rectangular pixels. We do however need to transpose the axial and coronal images to display them in the conventional orientation.

In each case we extract the relevant slice from the image volume, then superimpose the same slice from the masked mask. Finally we plot the line through the corner vertices to show all voxel edges. For the different orientations we need to extract different pairs from the (x,y,z) triples in the `corner_coords` array.

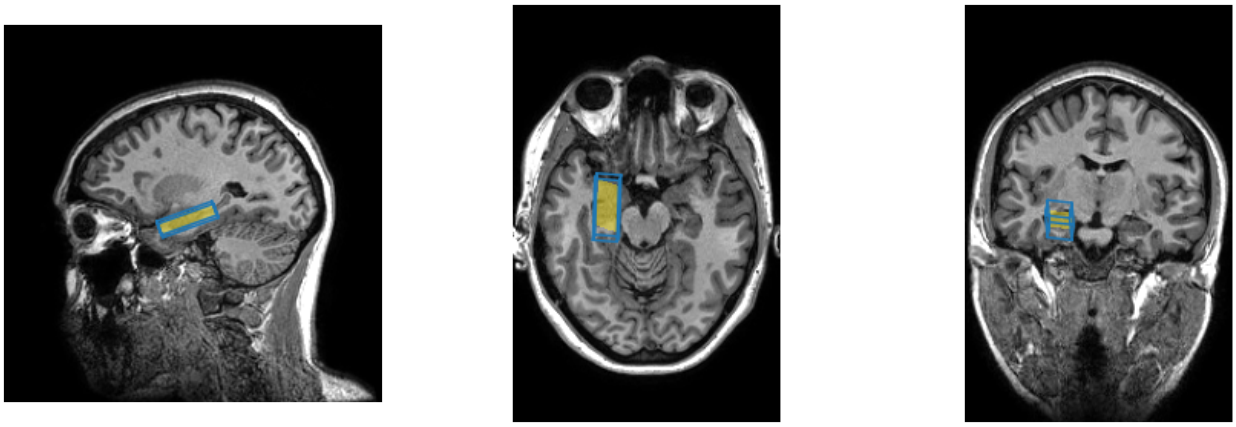
[7]:
```
fig, (ax_sag, ax_ax, ax_cor) = plt.subplots(1, 3, figsize=(15, 5))
ax_sag.axis('off')
ax_ax.axis('off')
ax_cor.axis('off')


ax_sag.imshow(image[voxel_centre_in_image[2]], cmap=plt.cm.gray, vmax=1400)
ax_ax.imshow(image[:, voxel_centre_in_image[1]].T, cmap=plt.cm.gray, vmax=1400)
ax_cor.imshow(image[:, :, voxel_centre_in_image[0]].T, cmap=plt.cm.gray, vmax=1400)

ax_sag.imshow(masked_mask[voxel_centre_in_image[2]], cmap=plt.cm.viridis_r, alpha=0.5)
ax_ax.imshow(masked_mask[:, voxel_centre_in_image[1]].T, cmap=plt.cm.viridis_r, alpha=0.
↪5)
ax_cor.imshow(masked_mask[:, :, voxel_centre_in_image[0]].T, cmap=plt.cm.viridis_r,␣
↪alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
ax_sag.plot(corner_coords[:, 0], corner_coords[:, 1])
ax_ax.plot(corner_coords[:, 2], corner_coords[:, 0])
ax_cor.plot(corner_coords[:, 2], corner_coords[:, 1])
```

[7]:
```
[<matplotlib.lines.Line2D at 0x7fb13e6a8a10>]
```



Because the voxel in this case is double oblique it can be difficult to get a good sense of its orientation from either the outline or the mask through a single slice. To help with this problem we can use the `resample()` function to adjust the orientation of the image so that the image direction vectors align with those of the spectroscopy voxel.

Because the image is sagittal and the spectroscopy voxel is axial, it can be complicated to work out which axes to choose. Fortunately, the `MRSData` class provides some helper properties to find the axes which are closest to the axial/sagittal/coronal directions, so we don't need to work out whether we want the slice/column/row directions. The only thing that can trip us up is that the image is acquired with a negative vector in the axial direction, whereas the spectroscopy voxel has a positive one - we solve this by setting a negative voxel size in that direction.

[8]:
```
res_image = image.resample(data.coronal_vector,
                           data.axial_vector,
                           image.shape,
                           image.centre,
                           (1, -1, 1),
                           )
```

We could also resample the mask but that would introduce partial volume effects, so instead we create a new mask from the resampled image. We also get the centre of the voxel in this new, resampled image volume.

[9]:
```
mask_res = suspect.image.create_mask(data, res_image)
masked_mask_res = np.ma.masked_where(mask_res == False, mask_res)

voxel_centre_in_res_image = res_image.from_scanner(data.position).round().astype(int)
print(voxel_centre_in_res_image)
```

```
[123 133 105]
```

We now replot the same 3 orthogonal slices as before, but using the resampled image, which puts the spectroscopy voxel perfectly aligned with the image planes.

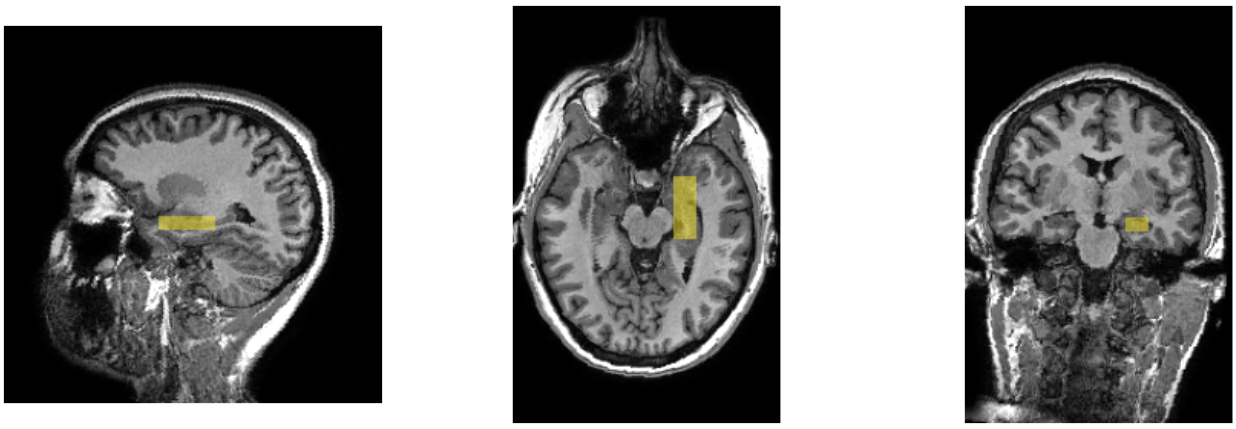
[10]:
```
fig, (ax_sag, ax_ax, ax_cor) = plt.subplots(1, 3, figsize=(15, 5))
ax_sag.axis('off')
ax_ax.axis('off')
```

(continues on next page)

(continued from previous page)

```
ax_cor.axis('off')

ax_sag.imshow(res_image[voxel_centre_in_res_image[2]], cmap=plt.cm.gray, vmax=1400)
ax_ax.imshow(res_image[:, voxel_centre_in_res_image[1]].T, cmap=plt.cm.gray, vmax=1400)
ax_cor.imshow(res_image[:, :, voxel_centre_in_res_image[0]].T, cmap=plt.cm.gray,␣
↪vmax=1400)

ax_sag.imshow(masked_mask_res[voxel_centre_in_res_image[2]], cmap=plt.cm.viridis_r,␣
↪alpha=0.5)
ax_ax.imshow(masked_mask_res[:, voxel_centre_in_res_image[1]].T, cmap=plt.cm.viridis_r,␣
↪alpha=0.5)
ax_cor.imshow(masked_mask_res[:, :, voxel_centre_in_res_image[0]].T, cmap=plt.cm.viridis_
↪r, alpha=0.5)
```

[10]: `<matplotlib.image.AxesImage at 0x7fb13da85650>`



We hope you have found this brief exploration of how to combine MRS and MRI data together. If you have questions, or suggestions on how to improve this guide, please raise an issue on the Suspect GitHub page, or post something in the MRSHub forum.

Co-registering Images

Learn how to combine anatomical scans with your MRS voxels

CHAPTER

THREE

THE CONSENSUS PROCESSING PIPELINE

As part of the 2020 NMR in Biomed Special Edition on Spectroscopy, Near et al. wrote a paper giving the consensus opinion on the post-acquisition processing steps, at least for the single voxel case:

Near, J., Harris, A. D., Juchem, C., Kreis, R., Marjańska, M., Öz, G., et al. (2020). Preprocessing, analysis and quantification in single-voxel magnetic resonance spectroscopy: experts' consensus recommendations. NMR in Biomedicine, 29, 323–23. http://doi.org/10.1002/nbm.4257

This is a highly worthwhile read for any spectroscopist, with excellent detail on almost every component of the process, with the one notable exception of channel combination.

Of course it is possible to perform all the processing steps from within Suspect, here we provide a Jupyter notebook with the complete consensus pipeline implemented.

3.1 OpenMRSLab SVS Demo Notebook

This notebook is designed to showcase some of the features provided by OpenMRSLab and the Suspect package for handling single voxel MRS data.

[1]:
```
import numpy as np
import nipype
from nipype.interfaces import fsl
import suspect
import os

import matplotlib.pyplot as plt
```

[2]:
```
%matplotlib notebook
```

In this example we are working with some SVS PRESS data in the Siemens twix format. We have both a standard water suppressed file and a water reference, and there is also a Nifti file with a T1 MPRAGE image.

[3]:
```
data_file = "meas_MID00062_FID19147_svs_se_30_PCC.dat"
wref_file = "meas_MID00063_FID19148_svs_se_30_PCC_wref.dat"
t1_file = os.path.abspath("t1.nii")
```

[4]:
```
t1 = suspect.image.load_nifti(t1_file)
data = suspect.io.load_twix(data_file)
wref = suspect.io.load_twix(wref_file)
```

Loading the twix data creates objects of the MRSData class. This class has various properties to allow us to probe the acquisition parameters, and also overrides the Python `__repr__()` function so that when we just print the `data` object, rather than a huge table of numbers we get a short representation with some key parameters included.

[5]:
```
print(data)
print(data.sw)
print(data.metadata)
print(data.shape)
```

```
<MRSBase instance f0=123.253243MHz TE=30.0ms dt=0.4167000000000007ms>
2399.808015358771
{'patient_name': 'xxxxxxxxxx', 'patient_id':
↪'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx', 'patient_birthdate': 'xxxxxxxx',
↪'exam_date': 'xxxxxx', 'exam_time': 'xxxxxx'}
(128, 2, 32, 2048)
```

As you can see from the `shape` property above, this dataset has 128 averages, 32 channels and 2048 ADC points in the FID, but there is an additional axis here with size 2. This is because with the VE software release Siemens uses a prospective frequency correction method where the excited signal from the WET water suppression pulses is collected and used to adjust the scanner frequency to correct for any instability in the signal. We don't want to use that data, so we extract only the other part of the signal, the part that we really care about.

[6]:
```
data = data[:, 1]
wref = wref[:, 1]
```

3.1.1 Channel Combination

Because this is multi-channel data, there is likely to be some level of correlation between some of the channels. We can check this by looking at the pure noise signal at the end of the FIDs. From the covariance matrix below we can see that there is indeed substantial correlation between various pairs of coils, and also variation in the magnitude of the signal from different coils.

[7]:
```
noise_points = 256
noise = data[:, :, -noise_points:]
noise = np.moveaxis(noise, -2, 0).reshape((32, -1))
plt.imshow(np.cov(noise).real)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[7]:
```
<matplotlib.image.AxesImage at 0x7fa4c2c11be0>
```

We can correct for this and "whiten" the noise using the method described in Rodgers, C. T., & Robson, M. D. (2010). Receive array magnetic resonance spectroscopy: Whitened singular value decomposition (WSVD) gives optimal Bayesian solution. Magnetic Resonance in Medicine, 63(4), 881–891. This creates a set of "virtual channels" as linear combinations of the real channels, producing zero correlation and equal magnitude of each channel.

[8]:
```
white_data = suspect.processing.channel_combination.whiten(data, noise)
white_wref = suspect.processing.channel_combination.whiten(wref, noise)
noise = white_data[:, :, -noise_points:]
noise = np.moveaxis(noise, -2, 0).reshape((32, -1))
plt.imshow(np.cov(noise).real)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[8]:
```
<matplotlib.image.AxesImage at 0x7fa4c2b2a9b0>
```

We then combine these virtual channels using the SVD weighting method proposed in the same paper. In this case we take the weightings for both datasets from the water reference signal for consistency, as the water reference has a greater overall SNR.

[9]:
```
channel_weights = suspect.processing.channel_combination.svd_weighting(np.mean(white_
↪wref, axis=0))
cc_data = suspect.processing.channel_combination.combine_channels(white_data, channel_
↪weights)
cc_wref = suspect.processing.channel_combination.combine_channels(white_wref, channel_
↪weights)
```

3.1.2 Frequency Drift Correction

Now it is time to check for frequency drift. In this demonstration we use a visualisation method borrowed from GANNET where we show the sequence of spectra as a heatmap. We can easily convert from a time domain FID representation of the data to spectra using the `spectrum()` method of the MRSData class. We then use the `slice_ppm()` method to index into the spectra in the range of frequencies we are interested in.

[10]:
```
spectra = cc_data.spectrum()
frequency_slice = spectra.slice_ppm(3.5, 1.9)
plt.imshow(spectra[:, frequency_slice].T.real, extent=[0, 128, 1.9, 3.5], aspect='auto')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[10]:
```
<matplotlib.image.AxesImage at 0x7fa4c2aa3518>
```

Despite the prospective frequency correction, a fairly pronounced drift in frequency is still visible over the 128 averages, this adds up to about 3Hz first spectrum to last. We can correct for that using Jamie Near's time domain Spectral Registration method. We use the first average in the list as the target and iterate over all averages, correcting them to maximum alignment with that target. Note that alternative methods such as Martin Wilson's RATS are also available. We also take the opportunity to make the same correction to the water reference data, although the drift in that case is likely minimal due to the reduced number of averages. Plotting the heatmap on the aligned data we can see that the drift is removed very effectively.

[11]:
```
sr_data = suspect.processing.frequency_correction.correct_frequency_and_phase(cc_data,␣
↪cc_data[0], method="sr")
sr_wref = suspect.processing.frequency_correction.correct_frequency_and_phase(cc_wref,␣
↪cc_wref[0], method="sr")
```

[12]:
```
sr_spectra = sr_data.spectrum()
frequency_slice = sr_spectra.slice_ppm(3.5, 1.9)
plt.imshow(sr_spectra[:, frequency_slice].T.real, extent=[0, 128, 1.9, 3.5], aspect='auto
↪')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[12]:
```
<matplotlib.image.AxesImage at 0x7fa4c228ee80>
```

With the frequency drift corrected, we can now combine the averages using a simple `mean()` function.

[13]:
```
ave_data = np.mean(sr_data, axis=0)
ave_wref = np.mean(sr_wref, axis=0)
```

3.1.3 Eddy Current Correction

The next step to consider is eddy current correction, which we analyse in the water reference data. We can very easily get the eddy current effect using standard NumPy functions.

[14]:
```
plt.plot(np.unwrap(np.angle(ave_wref)))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[14]:
```
[<matplotlib.lines.Line2D at 0x7fa4c21f7ba8>]
```

In this case we do see substantial eddy current effects, so we will calculate a correction term. Here we use a denoising function from Suspect to smooth out the curve, although we can't really do anything at the end of the acquisition window where the signal is pure noise, it doesn't matter what we do to the signal there anyway. Replotting the corrected water reference phase evolution shows no discernable eddy current effects.

[15]:
```
eddy_current = np.unwrap(np.angle(ave_wref))
ec_smooth = suspect.processing.denoising.sliding_gaussian(eddy_current, 32)
ecc = np.exp(-1j * ec_smooth)
```

[16]:
```
ec_data = ave_data * ecc
ec_wref = ave_wref * ecc
plt.plot(np.unwrap(np.angle(ec_wref)))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7fa4c21daa58>]
```

It is finally time to plot our spectrum. Again we make use of a helper method of the MRSData class to generate a frequency axis in ppm to match our spectral data.

[17]:
```
plt.plot(ec_data.frequency_axis_ppm(), ec_data.spectrum().real)
plt.xlim([5, 0])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[17]:
```
(5, 0)
```

3.1.4 Residual Water Removal

Oops! There is still a lot of residual water left in the signal, much bigger than the metabolite peaks. We could leave this to Tarquin to take out, but why not do it ourselves? We use the HSVD method to decompose the FID into exponential decay signals, then we select all those signals with a frequency below 60 Hz from centre, assuming those to be only parts of the water peak. We construct a new FID from those components which represents the pure residual water signal.

[18]:
```
components = suspect.processing.water_suppression.hsvd(ec_data, 30)
water_components = [component for component in components if component["frequency"] < 60]
water_fid = ec_data.inherit(suspect.processing.water_suppression.construct_fid(water_
↪components, ec_data.time_axis()))
plt.plot(ec_data.frequency_axis_ppm(), ec_data.spectrum().real)
plt.plot(water_fid.frequency_axis_ppm(), water_fid.spectrum().real)
plt.xlim([5, 0])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[18]:
```
(5, 0)
```

Now we can subtract out that water signal and at last plot our final spectrum.

[19]:
```
dry_data = ec_data - water_fid
plt.plot(dry_data.frequency_axis_ppm(), dry_data.spectrum().real)
plt.xlim(5, 0)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[19]:
```
(5, 0)
```

3.1.5 Absolute Quantification

Now before doing a fitting step, we need to make some calculations to get parameters to allow us to do absolute quantification. We define a little function using NiPype, which allows us to run FSL routines from Python, in this case to run the BET and FAST programs to remove the brain from our T1 image and classify it into grey matter, white matter and CSF.

[20]:
```
def classify_tissues(t1_file):
    """
    Given a NIFTI file containing a T1 head image, run the FSL tools BET and FAST
    to extract the brain and classify it into white matter, grey matter and CSF
    labels, then return 3 image volumes with the voxelwise probabilities of
    membership of each label.
    """
    workflow = nipype.Workflow(name="classify_tissues")
    bet = nipype.Node(fsl.BET(frac=0.5,
                              robust=True),
                      name="bet")
    fast = nipype.Node(fsl.FAST(output_type="NIFTI",
                                number_classes=3),
                       name="fast")
```

(continues on next page)

(continued from previous page)

```
    workflow.connect([(bet, fast, [("out_file", "in_files")])])
    bet.inputs.in_file = os.path.abspath(t1_file)
    result = workflow.run()
    for node in result.nodes():
        if node.name == "fast":
            wm = suspect.image.load_nifti(node.result.outputs.partial_volume_files[2])
            gm = suspect.image.load_nifti(node.result.outputs.partial_volume_files[1])
            csf = suspect.image.load_nifti(node.result.outputs.partial_volume_files[0])
    return wm, gm, csf
```

[21]:
```
wm, gm, csf = classify_tissues(t1_file)
```

```
211201-10:40:01,743 workflow INFO:
         Workflow classify_tissues settings: ['check', 'execution', 'logging',
↪'monitoring']
211201-10:40:01,804 workflow INFO:
         Running serially.
211201-10:40:01,807 workflow INFO:
         [Node] Setting-up "classify_tissues.bet" in "/tmp/tmpgf55rwe_/classify_tissues/
↪bet".
211201-10:40:01,819 workflow INFO:
         [Node] Running "bet" ("nipype.interfaces.fsl.preprocess.BET"), a CommandLine␣
↪Interface with command:
bet /home/jovyan/work/consensus/t1.nii /tmp/tmpgf55rwe_/classify_tissues/bet/t1_brain.
↪nii.gz -f 0.50 -R
211201-10:40:18,615 workflow INFO:
         [Node] Finished "classify_tissues.bet".
211201-10:40:18,619 workflow INFO:
         [Node] Setting-up "classify_tissues.fast" in "/tmp/tmpx4elpz4z/classify_tissues/
↪fast".
211201-10:40:18,636 workflow INFO:
         [Node] Running "fast" ("nipype.interfaces.fsl.preprocess.FAST"), a CommandLine␣
↪Interface with command:
fast -n 3 -S 1 /tmp/tmpx4elpz4z/classify_tissues/fast/t1_brain.nii.gz
211201-10:45:28,83 workflow INFO:
         [Node] Finished "classify_tissues.fast".
```

Now we create a mask of the image voxels which are inside the PRESS box. From our tissue label maps we can work out the fraction of each tissue type found in the voxel. Luckily the fractions all add up to 1. We then use Suspect to calculate the molar concentration factor which converts the ratio of metabolite to water peak to a metabolite molar concentration.

[22]:
```
voxel_mask = suspect.image.create_mask(data, t1)
```

[23]:
```
voxel_volume = np.sum(voxel_mask)
f_wm = np.sum(wm * voxel_mask) / voxel_volume
f_gm = np.sum(gm * voxel_mask) / voxel_volume
f_csf = np.sum(csf * voxel_mask) / voxel_volume
```

[24]:
```
print(f_csf + f_wm + f_gm)
```

```
1.0000000001455192
```

[25]:
```
aq_factor = suspect.fitting.molar_concentration_factor(f_wm, f_gm, f_csf, data.te, data.
↪tr)
print(aq_factor)
```

```
34573.68835797036
```

3.1.6 Fitting

We can now pass the data, water reference and absolute quantification factor into the Tarquin fitting program, which will then return the fitting results back to us. We get Tarquin's processed version of the data, its fit and baseline estimates, as well as lines for each metabolite in the basis set.

[26]:
```
fit = suspect.fitting.tarquin.process(dry_data, ec_wref, aq_factor=aq_factor)
```

[27]:
```
plt.plot(fit["plots"]["data"].frequency_axis_ppm(), fit["plots"]["data"].real)
plt.plot(fit["plots"]["data"].frequency_axis_ppm(), fit["plots"]["fit"].real + fit["plots
↪"]["baseline"].real)
plt.plot(fit["plots"]["data"].frequency_axis_ppm(), fit["plots"]["metabolites"]["NAA"].
↪real - 25)
plt.plot(fit["plots"]["data"].frequency_axis_ppm(), fit["plots"]["metabolites"]["Cr"].
↪real + fit["plots"]["metabolites"]["PCr"].real - 50)
plt.plot(fit["plots"]["data"].frequency_axis_ppm(), fit["plots"]["metabolites"]["GPC"].
↪real - 75)
plt.xlim([5, 0.0])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[27]:
```
(5, 0.0)
```

Of course, we also get the numerical information about the metabolite concentrations and standard deviations, and we also get the fit quality information provided by Tarquin.

[28]:
```
fit["metabolite_fits"]
```

[28]:
```
{'-CrCH2': {'concentration': '0.000', 'sd': 'inf'},
 'Ala': {'concentration': '0.000', 'sd': 'inf'},
 'Asp': {'concentration': '5.730', 'sd': '8.240'},
 'Cr': {'concentration': '6.358', 'sd': '3.897'},
 'GABA': {'concentration': '2.366', 'sd': '21.69'},
 'GPC': {'concentration': '1.765', 'sd': '3.229'},
 'Glc': {'concentration': '0.5625', 'sd': '35.49'},
 'Gln': {'concentration': '0.000', 'sd': 'inf'},
 'Glth': {'concentration': '2.740', 'sd': '6.444'},
 'Glu': {'concentration': '11.38', 'sd': '3.326'},
 'Ins': {'concentration': '6.880', 'sd': '2.097'},
 'Lac': {'concentration': '0.2857', 'sd': '37.89'},
 'Lip09': {'concentration': '1.918', 'sd': '58.48'},
 'Lip13a': {'concentration': '5.336', 'sd': '225.0'},
 'Lip13b': {'concentration': '0.000', 'sd': 'inf'},
 'Lip20': {'concentration': '6.858', 'sd': '19.72'},
 'MM09': {'concentration': '6.182', 'sd': '20.49'},
 'MM12': {'concentration': '2.004', 'sd': '293.0'},
```

(continues on next page)

(continued from previous page)

```
 'MM14': {'concentration': '3.214', 'sd': '562.0'},
 'MM17': {'concentration': '3.322', 'sd': '220.2'},
 'MM20': {'concentration': '11.08', 'sd': '14.96'},
 'NAA': {'concentration': '10.36', 'sd': '2.432'},
 'NAAG': {'concentration': '1.819', 'sd': '21.47'},
 'PCh': {'concentration': '0.000', 'sd': 'inf'},
 'PCr': {'concentration': '3.199', 'sd': '6.584'},
 'Scyllo': {'concentration': '0.2962', 'sd': '10.38'},
 'Tau': {'concentration': '2.868', 'sd': '9.566'},
 'TNAA': {'concentration': '12.18', 'sd': '1.534'},
 'TCho': {'concentration': '1.765', 'sd': '1.311'},
 'TCr': {'concentration': '9.557', 'sd': '0.9290'},
 'Glx': {'concentration': '11.38', 'sd': '3.144'},
 'TLM09': {'concentration': '8.100', 'sd': '7.178'},
 'TLM13': {'concentration': '10.55', 'sd': '19.81'},
 'TLM20': {'concentration': '17.94', 'sd': '8.427'}}
```

[29]:
```
print(fit["quality"])
```

```
{'Metab FWHM (PPM)': 0.02614, 'Metab FWHM (Hz)': 3.222, 'SNR residual': 49.81, 'SNR max':
↪ 171.3, 'Q': 3.439}
```

3.1.7 Voxel Plotting

As a final demonstration, we render the PRESS box on top of the anatomical image. Both MRS and MRI objects have affine transforms relating their internal grids to the scanner coordinate system, so we can easily convert the `position` property of `data` (the centre of the PRESS box) into a voxel of the T1 image, telling us which slice passes through the centre of the MRS voxel. We then define a square of coordinates around the edges of the MRS voxel which are converted first from MRS to scanner coordinates, then into T1 coordinates, where they are then ready for plotting on the displayed image.

[30]:
```
voxel_centre_index = t1.from_scanner(*data.position).round().astype(int)
corner_coords_voxel = [[0, -0.5, -0.5],
                       [0, 0.5, -0.5],
                       [0, 0.5, 0.5],
                       [0, -0.5, 0.5],
                       [0, -0.5, -0.5]]
corner_coords = np.array([t1.from_scanner(*data.to_scanner(*coord)) for coord in corner_
↪coords_voxel])
```

[31]:
```
plt.imshow(t1[voxel_centre_index[2]], cmap=plt.cm.gray)
plt.plot(corner_coords[:, 0], corner_coords[:, 1], 'yellow')
plt.xlim([0, t1.shape[2] - 1])
plt.ylim([t1.shape[1] - 1, 0])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[31]:
```
(255, 0)
```

```
[ ]:
```

OpenMRSLab SVS Demo Notebook

CHAPTER

FOUR

API REFERENCE

4.1 Suspect API Reference

suspect.**adjust_frequency**(*data*, *frequency_shift*)

Adjust the centre frequency of an MRSBase object.

Parameters

frequency_shift (*float*) – The amount to shift the frequency, in Hertz.

Returns

out – Frequency adjusted FID

Return type

MRSData

suspect.**adjust_phase**(*data*, *zero_phase*, *first_phase=0*, *fixed_frequency=0*)

Adjust the phase of an MRSBase object

Parameters

- **data** (*MRSSpectrum*) – The MRSSpectrum object to be phased
- **zero_phase** (*scalar*) – The change to the zero order phase, in radians
- **first_phase** (*scalar, optional*) – The change to the first order phase, in radians per Hz
- **fixed_frequency** (*scalar, optional*) – The frequency, in Hz, which is unchanged by the first order phase shift

Returns

out – A new MRSSpectrum object with adjusted phase.

Return type

MRSSpectrum

4.2 MRSData Reference

class suspect.**MRSData**(*input_array*, *dt*, *f0*, *te=30*, *tr=- 1*, *ppm0=4.7*, *voxel_dimensions=(10, 10, 10)*, *transform=None*, *metadata=None*)

MRS data in the time domain.

adjust_frequency(*frequency_shift*)

Adjust the centre frequency of the signal.

Refer to suspect.adjust_frequency for full documentation.

Parameters
frequency_shift (*float*) – The amount to shift the frequency, in Hertz.

Returns
out – Frequency adjusted FID

Return type
MRSData

See also:

suspect.adjust_frequency
equivalent function

adjust_phase(*zero_phase*, *first_phase=0.0*, *fixed_frequency=0.0*)

Adjust the phases of the signal.

Refer to suspect.adjust_phase for full documentation.

Parameters

- **zero_phase** (*float*) – The zero order phase shift in radians
- **first_phase** (*float*) – The first order phase shift in radians per Hertz
- **fixed_frequency** (*float*) – The frequency at which the first order phase shift is zero

Returns
out – Phase adjusted FID

Return type
MRSData

See also:

suspect.adjust_phase
equivalent function

fid()

Returns itself. This is useful when you have either a spectrum or an FID, but want an FID

Returns
The called MRSData object

Return type
MRSData

spectrum()

Returns
The Fourier-transformed and shifted data, represented as a spectrum

Return type
MRSSpectrum

4.3 Frequency Correction API Reference

suspect.processing.frequency_correction.**correct_frequency_and_phase**(*data*, *target*, *method='sr'*, *axis=- 1*, ***kwargs*)

Interface to frequency and phase correction algorithms, but returning the corrected data rather than the calculated shifts. Can be applied to multi-dimensional data to align e.g. a sequence of spectra at once.

Parameters

- **data** (*MRSBase*) – The data to be corrected.
- **target** (*MRSBase*) – The reference spectrum to which data will be aligned.
- **method** (*str, optional*) – The correction method to be used. Should be one of:
  - 'sr' Time-domain Spectral Registration - see *spectral_registration()*
  - 'rats' RATS (Robust Alignment to a Target Spectrum) - see *rats()*
  - 'rwa' Residual Water Alignment - see *residual_water_alignment()*
  - custom A callable object, see below
- **axis** (*int or None, optional*) – The axis defining the spectral dimension.
- **kwargs** – Arguments to be passed through to the method function.

Returns
The data with corrected frequency and phase.

Return type
MRSBase

Notes

Custom Frequency/Phase Corrections

If you have an alternative method for calculating frequency and phase shifts then you can simply pass a callable to the `method` parameter.

The callable is called as `method(moving_data, target_data, **kwargs)` where *kwargs ``corresponds to any other parameters that may be passed in such as `initial_guess*. The method shall return a 2-tuple of floats containing the frequency shift in Hertz and the phase shift in radians between the first passed data and the second. Note that it is the measured frequency and phase shifts that should be returned, they will be negated by this function to correct the spectrum to the target.

suspect.processing.frequency_correction.**rats**(*data*, *target*, *initial_guess=(0.0, 0.0)*, *frequency_range=None*, *baseline_order=2*, ***kwargs*)

Uses the RATS (Robust Alignment to a Target Spectrum)[1] to calculate the frequency and phase shifts between

[1] Wilson, M. (2018). Robust retrospective frequency and phase correction for single-voxel MR spectroscopy. Magnetic Resonance in Medicine, 81(5), 2878–2886. http://doi.org/10.1002/mrm.27605

the input data and a reference spectrum. RATS operates in the frequency domain and the frequencies used to align the spectra can be specified to exclude regions where the spectra differ.

Parameters

- **data** (MRSData) – The data to be aligned to the target
- **target** (MRSData) – The target data to which the moving data will be aligned
- **initial_guess** (*tuple*) – A 2-tuple of frequency and phase shifts at which the optimisation routine will start searching. See below for more information.
- **frequency_range** (*tuple, slice or ndarray*) – The frequency range can be specified in multiple different ways: a 2-tuple containing low and high frequency cut-offs in Hertz for the comparison, as a slice object into the spectrum (for use with the slice_ppm() function, or as an array of weights to apply to the spectrum.
- **baseline_order** (*int*) – The order of the polynomial baseline.

Returns

- **frequency_shift** (*float*) – The estimated frequency shift in Hz.
- **phase_shift** (*float*) – The estimated phase shift in radians.

Notes

Experimentally, I have found the RATS method to sometimes get stuck in local minima where the frequency shift is larger than about 6Hz. Although this can be addressed via the *initial_guess* parameter, when batch processing a sequence of spectra experiencing substantial drift, there is no single good value of *initial_guess*. In order to address this issue, this function begins by using a coarse grid based search to evaluate frequencies every 2Hz from 20Hz above the *initial_guess* frequency to 20Hz below. It then picks from those the frequency which gives best alignment with the target data as a starting point for the optimisation search. So far this has been very effective, but in case of discovering any problems, please raise an issue at http://github.com/openmrslab/suspect.

References

suspect.processing.frequency_correction.**residual_water_alignment**(*data*)

Parameters
data –

suspect.processing.frequency_correction.**spectral_registration**(*data*, *target*, *initial_guess=(0.0, 0.0)*, *frequency_range=None*, ***kwargs*)

Performs the spectral registration method[2] to calculate the frequency and phase shifts between the input data and the reference spectrum target. The frequency range over which the two spectra are compared can be specified to exclude regions where the spectra differ.

Parameters

- **data** (MRSData) –
- **target** (MRSData) –
- **initial_guess** (*tuple*) –

[2] Near, J., Edden, R., Evans, C. J., Paquin, R., Harris, A., & Jezzard, P. (2014). Frequency and phase drift correction of magnetic resonance spectroscopy data by spectral registration in the time domain. Magnetic Resonance in Medicine, 73(1), 44–50. http://doi.org/10.1002/mrm.25094

- **frequency_range** (*tuple, slice or ndarray*) – The frequency range can be specified in multiple different ways: a 2-tuple containing low and high frequency cut-offs in Hertz for the comparison, as a slice object into the spectrum (for use with the slice_ppm() function, or as an array of weights to apply to the spectrum.

Returns

- **frequency_shift** (*float*) – The estimated frequency shift in Hz.
- **phase_shift** (*float*) – The estimated phase shift in radians.

References

4.4 suspect.fitting API Reference

suspect.fitting.**attenuation_scaling_factor**(*t1*, *t2*, *te*, *tr*)

Calculates the expected attenuation in an MRS signal arising due to T1 and T2 effects, according to the equation: $e^{-\frac{TE}{T2}}(1 - e^{-\frac{TR}{T1}})$

Parameters

- **t1** – The T1 of the tissue
- **t2** – The T2 of the tissue
- **te** – The echo time of the sequence
- **tr** – The repetition time of the sequence

Returns
The attenuation factor in the observed signal expected due to T1 and T2 effects

Return type
float

Notes

This function uses the simplified form of the equation, assuming that TE is much shorter than T1 and so the sequence behaves roughly as a 90 pulse followed by a readout, any refocussing pulses do not substantially change the longitudinal magnetisation. If this is not the case then the details of the sequence will be important and a more precise form of this calculation will be required.

suspect.fitting.**molar_concentration_factor**(*f_wm*, *f_gm*, *f_csf*, *te*, *tr*, *tissue_params=None*)

Calculate the scaling factor necessary to obtain molar metabolite concentrations from a ratio of metabolite to water peak amplitude.

Parameters

- **f_wm** (*float*) – The fraction of the voxel containing white matter
- **f_gm** (*float*) – The fraction of the voxel containing grey matter
- **f_csf** (*float*) – The fraction of the voxel containing CSF
- **te** (*float*) – The echo time of the sequence
- **tr** (*float*) – The repetition time of the sequence
- **tissue_params** (*dict, optional*) – User supplied values for tissue MR properties including T1 and T2

Returns
The scaling factor to convert metabolite to water ratios to molar concentrations.

Return type
float

Notes

The calculation used here follows the form derived in[1].

There are various parameters required for these calculations which can vary, for example with field strength or other conditions. Default values are provided in Suspect (tissue water concentrations are drawn from Gasparovic 2006, relaxation times from Gussew 2012), but they can be overridden by passing in alternative values in the tissue_parameters argument. The relevant parameters are:

Key	Description	Default Value
beta_wm	the water density of white matter	0.65
beta_gm	the water density of grey matter	0.78
beta_csf	the water density of CSF	0.97
h2o_t1_wm	the T1 of water in white matter	1080ms
h2o_t1_gm	the T1 of water in grey matter	1820ms
h2o_t1_csf	the T1 of water in CSF	4160ms
h2o_t2_wm	the T2 of water in white matter	70ms
h2o_t2_gm	the T2 of water in grey matter	100ms
h2o_t2_csf	the T2 of water in CSF	500ms
met_t1	the T1 of metabolites in brain tissue	1400ms
met_t2	the T2 of metabolites in brain tissue	200ms

References

suspect.fitting.tarquin.**process**(*data*, *wref=None*, *aq_factor=None*, *options={}*)

Runs the Tarquin basis set fitting program to determine metabolite concentrations.

Parameters

- **data** (MRSData) – The water suppressed FID data to be fitted.
- **wref** (MRSData) – Optional water reference file for concentration scaling.
- **aq_factor** (*float*) – Absolute quantification factor.
- **options** (*dict*) – Set of Tarquin parameters to override.

Returns
Output from running Tarquin on the data

Return type
dict

class suspect.fitting.singlet.**GaussianPeak**(*name*, *amplitude=1*, *frequency=0*, *phase='0'*, *fwhm=20*)

Class to represent a Gaussian peak for fitting.

The Gaussian peak is parameterised by 4 values: amplitude, frequency, phase, and FWHM (full width at half maximum).

[1] Near, J., Harris, A. D., Juchem, C., Kreis, R., Marjańska, M., Öz, G., et al. (2020). Preprocessing, analysis and quantification in single-voxel magnetic resonance spectroscopy: experts' consensus recommendations. NMR in Biomedicine, 29, 323–23. http://doi.org/10.1002/nbm.4257

Each parameter can be specified in three different ways: 1. Passing a numeric value sets the initial guess for that parameter 2. Passing a string of a number fixes that parameter to that value 3. Passing a dictionary allows setting any of the constraints supported

by the underlying LMFit parameter: value, min, max, vary, and expr

By default the phase of the peak will be fixed at 0 and the amplitude and FWHM will be constrained to be bigger than 0 and 1Hz respectively.

Parameters

- **name** – The name of the peak
- **amplitude** – The amplitude (area) of the peak
- **frequency** – The frequency of the peak in Hertz
- **phase** – The phase of the peak in radians
- **fwhm** – The full width at half maximum of the peak in Hertz

class suspect.fitting.singlet.**Model**(*peak_models*, *phase0=0*, *phase1='0'*)

A model of an MRS FID signal which can be fitted to data.

This model is created by passing a set of individual peak models, to which it then appends a phase model. By default the first order phase is constrained to be 0.

Parameters

- **peak_models** – The descriptions of the peaks making up the model.
- **phase0** – The estimated zero order phase in radians.
- **phase1** – The estimated first order phase in radians per Hz.

fit(*data*, *baseline_points=4*)

Perform a fit of the model to an FID.

Parameters

- **data** – The time domain data to be fitted.
- **baseline_points** – The first baseline_points of the FID will be ignored in the fit.

Return type
ModelResult

classmethod from_dict(*model_dict*)

Create a model from a dict.

Parameters
model_dict – dict describing the model.

Returns
The specified model ready for fitting.

Return type
Model

4.5 Changelog

Please see the changelog.

PYTHON MODULE INDEX

S

suspect, 41
suspect.fitting, 45
suspect.fitting.singlet, 46
suspect.fitting.tarquin, 46
suspect.processing.frequency_correction, 43

INDEX

A

adjust_frequency() (*in module suspect*), 41
adjust_frequency() (*suspect.MRSData method*), 42
adjust_phase() (*in module suspect*), 41
adjust_phase() (*suspect.MRSData method*), 42
attenuation_scaling_factor() (*in module suspect.fitting*), 45

C

correct_frequency_and_phase() (*in module suspect.processing.frequency_correction*), 43

F

fid() (*suspect.MRSData method*), 42
fit() (*suspect.fitting.singlet.Model method*), 47
from_dict() (*suspect.fitting.singlet.Model class method*), 47

G

GaussianPeak (*class in suspect.fitting.singlet*), 46

M

Model (*class in suspect.fitting.singlet*), 47
module
    suspect, 41, 42
    suspect.fitting, 45
    suspect.fitting.singlet, 46
    suspect.fitting.tarquin, 46
    suspect.processing.frequency_correction, 43
molar_concentration_factor() (*in module suspect.fitting*), 45
MRSData (*class in suspect*), 42

P

process() (*in module suspect.fitting.tarquin*), 46

R

rats() (*in module suspect.processing.frequency_correction*), 43
residual_water_alignment() (*in module suspect.processing.frequency_correction*), 44

S

spectral_registration() (*in module suspect.processing.frequency_correction*), 44
spectrum() (*suspect.MRSData method*), 42
suspect
    module, 41, 42
suspect.fitting
    module, 45
suspect.fitting.singlet
    module, 46
suspect.fitting.tarquin
    module, 46
suspect.processing.frequency_correction
    module, 43