gcmstools Documentation

Release 0.2.0

Ryan Nelson

May 14, 2015

CONTENTS

CHAPTER

ONE

GETTING STARTED

gcmstools is a Python package that reads some GCMS file formats and does simple fitting and calibration. The source code for this project can be found on GitHub. If you are reading this in PDF format, there is online documentation as well.

This user guide is broken into a few sections.

1. *Installation*: Information about getting a Python installation up and running and installing *gcmstools*.
2. *Basic Setup*: Go here first to learn about the basic setup and to get some sample data. This is necessary to follow this documentation.
3. *File Types*: This covers the supported file types, their properties, and some simple plotting.
4. *Referenceing and Fitting*: Incorporate reference data into your GCMS dataset and use this to fit the data.
5. *Data Storage*: Create a storage container for your processed GCMS datasets. These containers are necessary for the next step in the process.
6. *Calibration*: Generate calibration curves and use this information to extract concentration information from you sample datasets.
7. *Isotopic Analysis*: Calculate the isotopic distribution of a sample using a series of reference mass spectra.
8. *Batch Processing*: A simple function for automating this entire process is presented. You can skip to this section if all you want to do is automate some data extractions. It is not necessary to understand the basics of data manipulation, calibration, etc.
9. *Appendix A*: Command line basics. The examples presented in this document require a basic working knowledge of using a command-line terminal interface and of running Python commands from an IPython interpreter. This section covers some of the basics.
10. *Appendix B*: Examples. Basic data extraction and plotting examples are presented. Check out this section if you want some ideas about processing, plotting, or extracting data.

CHAPTER

TWO

INSTALLATION

Gcmstools requires Python and a number of third-party packages. Below is a complete list of packages and minium versions:

- Python >=3.4 (2.x versions not supported any longer)
- Pip >=6.0.6 (might be part of newer Python releases)
- Setuptools >=11.3.1
- Numpy >=1.9.1
- Matplotlib >= 1.4.2
- Pandas >=0.15.2
- IPython >=2.3.1
- PyTables >=3.1.1
- Scipy >=0.14.0
- Sphinx >=1.2.2 (Optional for documentation.)
  - numfig is a Sphinx extenstion that is needed to autonumber figures references in the docuementation.

IPython is also a very useful advanced interactive Python interpreter. Examples in this documentation assume that you are using this environment. See the *IPython* section of *Appendix A* for more details.

2.1 Python

The most convenient installation method for third-party Python packages is the all-in-one Anaconda Python distribution. It combines a large number of Python packages for scientific data analysis and a program (`conda`) for managing package updates, in addition to many other advanced features. The Anaconda developers (Continuum Analytics) provide a lot of useful documentation on installing Anaconda and using conda. There are other ways to install Python and the third-party packages, but for this documentation, it will be assumed that your are using Anaconda.

Note: On Mac/Linux, Python is already part of the operating system. Do not try to install these third-party packages into the builtin Python distribution unless you really know what you are doing. You might overwrite an important file, which can cause problems for your system. Confusion between the system and Anaconda Python installation is a common source of problems for beginners, so make sure that your Anaconda Python is "activated" before running the commands in this document. (See the Anaconda documentation for more information on the activation process.)

Learning the usage of all of these Python packages is far beyond the scope of this document. However, excellent documentation for most of the packages as well as full tutorials are easily discovered.

2.2 gcmstools

There are three installation options for *gcmstools*: 1) install using conda (recommended), 2) install using `pip`, or 3) install the most recent development version using `git`.

Option 1 (recommended)

If you are using the Anaconda Python distribution, you can use `conda` to install the most recent distribution of *gcmstools* from Binstar, Continuum's package repository.

```
home>$ conda install -c https://conda.binstar.org/rnelsonchem gcmstools
```

To uninstall *gcmstools* from a `conda` environment, use the following command:

```
home>$ conda remove gcmstools
```

Option 2

gcmstools can also be installed from the official Python packaging site, PyPI, using the standard Python installer script `pip`. This installation method will work fine if you have the additional dependencies installed; otherwise, you may have some problems depending on your platform.

```
home>$ pip install gcmstools
```

To uninstall *gcmstools* using `pip`:

```
home>$ pip uninstall gcmstools
```

Option 3

The development version of *gcmstools* is hosted on GitHub. To use this version, you must install the version-control software Git. *gcmstools* can then be downloaded and installed with one `pip` command.

```
home>$ pip install git+https://github.com/rnelsonchem/gcmstools.git
```

There are advantages and disadvantages with this approach. For example, the GitHub repo will always have the most recent updates and bug fixes; however, these new features might not be as well-tested so you might find new bugs.

To uninstall, use the method described in *Option 2*

CHAPTER

THREE

BASIC SETUP

Below are the instructions for setting up an environment that will let you work through this documentation.

3.1 The processing environment

In these examples, we will run *gcmstools* from a terminal IPython session in a folder "gcms", which is located in your home directory. Basic information about using the terminal and IPython is found in *Appendix A: Running Code Samples*.

```
home>$ cd gcms

gcms>$ ipython
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In :
```

3.2 Sample Data

A `zip archive containing example files` is provided in the online *gcmstools* documentation. These files can be downloaded into the current directory using the `get_sample_data` function.

```
In : from gcmstools.general import get_sample_data

In : get_sample_data()
```

CHAPTER

FOUR

GCMS FILETYPES

There are potentially many types of GCMS files; however, all file importing objects discussed in this section should have identical properties. This is important for later sections of the documentation, because fitting routines, etc., usually do not require a specific file type importer. All of the import objects are constructed with a single string input, which is the name of the file to process. This file name string can also contain path information if the file is not located in the current directory.

4.1 AIA Files

AIA, ANDI, or CDF are all related types of standard GCMS files that are derived from the Network Common Data Format (netCDF). They may have the file extension "AIA" or "CDF". This file type may not be the default for your instrument, so consult the documentation for your GCMS software to determine how to export your data in these formats.

To import this type of data, use the `AiaFile` object, which is located in the `gcmstools.filetypes` module.

```
In : from gcmstools.filetypes import AiaFile
```

Note: Currently, *gcmstools* can only process CDF version 3 files using `scipy.io.netcdf` library. Version 4 support could be available upon request.

4.1.1 Read an AIA File

File readers are imported from the `gcmstools.filetypes` module. In this example, we'll use the AIA file reader, `AiaFile`; however, the results should be identical with other readers. To read a file, you can create a new instance of this object with a filename given as a string.

```
In : from gcmstools.filetypes import AiaFile

In : data = AiaFile('datasample1.CDF')
Building: datasample1.CDF
```

The variable `data` now contains our processed GCMS data set. You can see its contents using *tab completion* in IPython.

```
In: data.<tab>
data.filename   data.intensity    data.tic    data.masses
data.filetype   data.int_extract  data.index  data.times
```

Most of these attributes are data that describe our dataset. You can inspect these attributes by typing the name at the IPython prompt.

```
In : data.times
Out:
array([0.08786667, ..., 49.8351])

In : data.tic
Out:
array([158521., ..., 0.])

In : data.filetype
Out: 'AiaFile'
```

This is a short description of these initial attributes:

- *filename*: String. This is the name of the file that you imported.
- *times*: A 1D Numpy array of the elution time points.
- *tic*: A 1D Numpy array of the total ion chromatogram (TIC) intensity values.
- *masses*: A 1D Numpy array the m/z values for the data collected by the MS.
- *intensity*: This is the 2D Numpy array of raw MS intensity data. The rows correspond to the times in the `times` array, and the columns correspond to the masses in the `masses` array. Shape(length of times, length of masses)
- *filetype*: String. This is the type of file importer that was used.

The *index* method is used for finding the indices from an array. Its usage is explained by example in *Appendix B*.

4.2 Simple plotting

Now that we've opened a GCMS data set, we can easily visualize these data using the plotting package Matplotlib. As an example, let's try plotting the total ion chromatogram. In this case, `data.times` will be our "x-axis" data, and `data.tic` will be our "y-axis" data.

```
In : import matplotlib.pyplot as plt

In : plt.plot(data.times, data.tic)
Out :
[<matplotlib.lines.Line2D at 0x7f34>]

In: plt.show()
```

This produces a interactive plot window shown in Figure 4.1. (This should happen fairly quickly. However, sometimes the plot window appears behind the other windows, which makes it seem like things are stuck. Be sure to scroll through your windows to find it.) The buttons at the top of the window give you some interactive control of the plot. See the Matplotlib documentation for more information.

One drawback here is that you have to type these commands every time you want to see this plot. Alternatively, you can put all of these commands into a text file and run it with Python directly. Copy the following code into a plain text file called "tic_plot.py". (See *Working with Text Files* for more information on making Python program files.) Note: It is common practice to do all imports at the top of a Python program. That way it is clear exactly what code is being brought into play.

```
import matplotlib.pyplot as plt
from gcmstools.filetypes import AiaFile

data = AiaFile('datasample1.CDF')
plt.plot(data.times, data.tic)
plt.show()
```

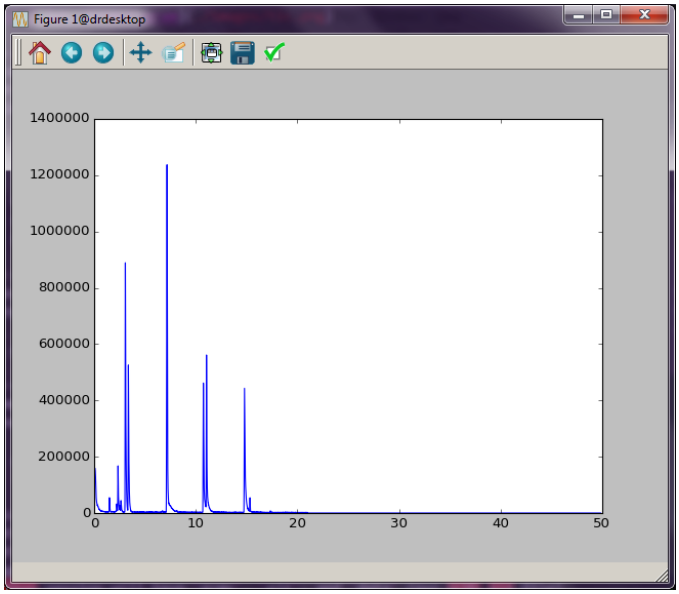

Figure 4.1: Total ion chromatogram.

Run this new file using the `python` command from the terminal. The plot window will appear, and you can interact with the data. However, you will not be able to work in the terminal again until you close this window.

```
gcms>$ python tic_plot.py
```

Alternatively, you can run this program directly from IPython. This has the advantage that once the window is closed, you are dropped back into an IPython session that "remembers" all of the variables and imports from your program file. See *Appendix A* for more information here.

```
In : %run tic_plot.py
```

4.3 Working with multiple data sets

In the example above, we opened one dataset into a variable called `data`. If you want to manipulate more than one data set, the procedure is the same, except that you will need to use different variable names for your other data sets. (Again, using AiaFile importer as an example, but this is not required.)

```
In : data2 = AiaFile('datasample2.CDF')
```

These two data sets can be plot together on the same figure by doing the following:

```
In : plt.plot(data.times, data.tic)
Out:
[<matplotlib.lines.Line2D at 0x7f34>]

In: plt.plot(data2.times, data2.tic)
Out:
[<matplotlib.lines.Line2D at 0x02e3>]

In: plt.show()
```

The window shown in Figure 4.2 should now appear. (There is a blue and green line here that are a little hard to see in this picture. Zoom in on the plot to see the differences.)

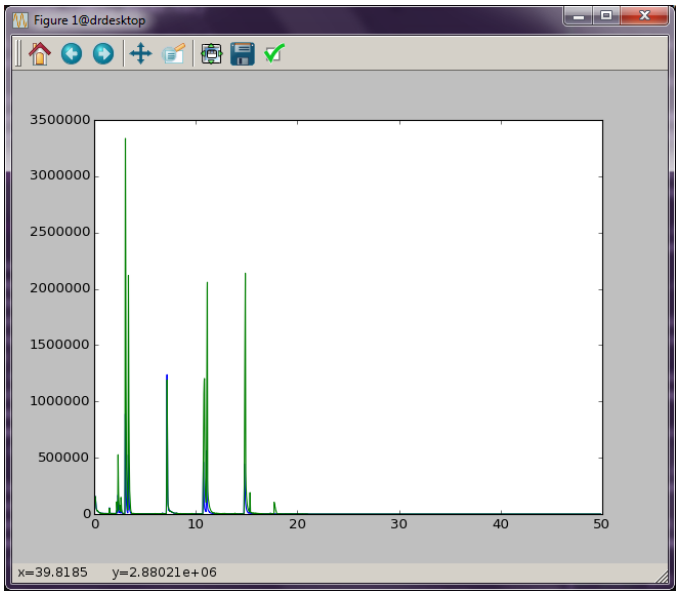

Figure 4.2: Two TIC plotted together.

CHAPTER

FIVE

REFERENCING AND FITTING

In order to properly integrate your GCMS data, you must assign reference information followed by running an appropriate fitting routine. This section discusses methods to accomplish this using *gcmstools*.

The general usage of the classes described in this section is identical. First, you will need one or more GCMS data objects, which you created in the *previous section*. Then you will create both reference and fitting object instances, which act as factories. You pass in a GcmsFile object, and the appropriate reference and fitting information will then be inserted into the object. Below is a pseudocode example that process:

```
# This will not work. See below for specific imports
In : from gcmstools.reference import FakeRef

In : from gcmstools.fitting import FakeFitter

In : ref = FakeRef('appropriate_reference_file')

In : fit = FakeFitter()

In : ref(data1)
Referencing: fakedata1

# Or pass in a list of many data objects
In : ref([data2, data3])
Referencing: fakedata2
Referencing: fakedata3

In : fit(data1)
Fitting: fakedata1

In : fit([data2, data3])
Fitting: fakedata2
Fitting: fakedata3

# all data sets now contain reference and fit information
```

Each reference and fitting type may append different types of information into your GCMS file object. In general, reference attributes will be preceded by "ref_", whereas fitting attributes will be preceded by "fit_".

One exception is that all referencing objects will add both a `ref_cpds` list and a `ref_meta` dictionary to the GCMS data object. The former is a list of the reference compound names that you define in the reference file, called "appropriate_reference_file" in our example. (More on this file below.) The latter contains metadata also extracted from the reference input file. When you pass your GCMS data object into the fitting object, an extracted integral will be added to that dictionary. These raw integrals are not useful on their own, but are used in the *Calibration and Concentration Determination* section later in this tutorial.

```
In : data1.ref_cpds
['ref_cpd_name1', 'ref_cpd_name2', 'ref_cpd_name3', ... ]

In : data1.ref_meta['ref_cpd_name1']['integral']
102237.01
```

5.1 Reference Files

In order to properly reference and fit your data set, you must prepare an appropriate reference input file. These files are simply plain text documents that describe the reference information for all of the compounds you would like to process. The structure of these files is very important. They are made up of a series of labels followed by the pertinent value. For example, all references must have a "NAME" label that provides the name of the reference compound. (It's probably best if this is concise.) In addition, if you are going to be integrating this compound, then you *must* include "START" and "STOP" values, which are the starting and ending elution times for the integration region. (Don't include the units.) Each reference compound *must* be separated by a blank line. Below is a truncated example for "benzene":

```
NAME:benzene
START:2.9
STOP:3.5
```

Additional labels can be added if you'd like to include extra metadata about your reference compound, such as retention index or molecular weight. For all reference file types, lines starting with # are treated as comment lines and ignored. This can be useful if you want to add some notes or to remove reference information without deleting them entirely.

Each reference/fitting type works with different reference files, so see the sections below for specifics.

5.2 Non-negative Least Squares

5.2.1 Collecting References for NNLS

The non-negative least squares (NNLS) fitting routine can use two different reference files. There are examples of both in the sample data directory: "ref_spec.txt" and "ref_spec2.MSL".

```
In : from gcmstools.general import get_sample_data

In : get_sample_data('ref_spec.txt')

In : get_sample_data('ref_spec2.MSL')
```

MSL files are typically autogenerated by external software, such as AMDIS or the NIST Spectral Database. The "txt" file was hand generated. In addition to the "NAME" (and potentially "START" and "STOP") label, these files have a section that starts with the label "NUM PEAKS" and is followed by the m/z and intensity information for the reference compound. As the MSL file is autogenerated, you will most like not have to adjust these data. However, you will have to add these data into the "txt" file by hand.

The "NUM PEAKS" label in a "txt" file is followed by (at least) two space-separated columns of MS data. The first column must be the m/z values, and the second column contains the associated intensity values. Intensities are normalized on import, so it is not necessary to do this by hand.

Remember, each reference compound *must* be separated by a blank line. Below is a small sample of one a "txt" file. An MSL file would differ only in the format of the "NUM PEAKS" section.

```
NAME:benzene
START:2.9
STOP:3.5
NUM PEAKS:
  36 1.82 18
  37 6.5 65
  38 7.38 74
  39 15.17 152
  49 6.91 69
.
.
.
```

The online MS repository massBank is a useful place to find these m/z and intensity values. The data from that site is already formated correctly for this file type.

5.2.2 Loading Reference Spectra

There are two objects located in `gcmstools.reference` for loading these reference data, `TxtReference` and `MslReferece`, which are used for "txt" and MSL reference files, respectively. In this example, we'll use `TxtReferece`, but the MSL version behaves in the same manner. The variable `data` refers to a GCMS file object that we created earlier.

```
In : from gcmstools.reference import TxtReference

In : ref = TxtReference('ref_specs.txt')

In : ref(data)
Referencing: datasample1.CDF

In : data.<tab>
data.filename     data.index        data.masses       data.ref_meta
data.filetype     data.ref_array    data.ref_type     data.tic
data.intensity    data.ref_cpds     data.times
```

Several new attributes have been added to our GCMS data object, all of which are prefixed with "ref_". Below is a short description of each. `ref_meta` and `ref_cpds` are described above.

- *ref_array*: A 2D Numpy array of the reference mass spectra. Shape(# of ref compounds, # of masses)
- *ref_type*: The name of the reference object type that was used to generate this information. (In this example, this would be "TxtReference".)

5.2.3 Fitting the data

A non-negative least squares fitting object, `Nnls`, is provided in the `gcmstools.fitting` module. To apply this fitting, simply call the fitting instance with a data object or list of data objects. This must be done after the data has been passed through a reference object.

```
In : from gcmstools.fitting import Nnls

In : fit = Nnls()

In : fit(data)
Fitting: datasample1.CDF
```

```
In : data.<tab>
data.filename     data.tic          data.fit_sim      data.ref_cpds
data.filetype     data.index        data.intensity    data.ref_meta
data.fit_coef     data.fit_csum     data.masses       data.ref_type
data.fit_type     data.ref_array    data.times
```

Again, several new attributes describing the fit, all starting with "fit_", have been added to our data set.

- *fit_type*: A string that names the fitting object used to generate this data. (In this case, it would be "Nnls".)
- *fit_coef*: A 2D Numpy array of the least squares coefficients at every time point. They do not correspond to proper integrations, so they should be used with caution. An example using these values to simulate a MS spectrum is shown in *Appendix B*.
- *fit_sim*: A 2D numpy array of simulated GCMS curves that were generated from the fit. Shape(# of time points, # of reference compounds)
- *fit_csum*: A 2D numpy array that is the cumulative summation of *fit_sim* along the time axis, so it has the same shape as that array. An integral of a particular region can be obtained by determining the difference between any two points along the time dimension in this array. However, the *calibration object* automatically handles this integration, so you shouldn't need to do integrations in this manner.

As stated above, the integrals for the reference compounds have also been added to the `ref_meta` dictionaries.

5.2.4 Plotting the Fit

You can do a quick visual check of the fits using Matplotlib. More advanced examples are presented in *Appendix B*. The output of the commands below is shown in Figure 5.1.

```
In : import matplotlib.pyplot as plt

In : plt.plot(data.times, data.tic, 'k-', lw=1.5)
Out: [<matplotlib.lines.Line2D at 0x7f9b2905df60>]

In : plt.plot(data.times, data.fit_sim)
Out:
[<matplotlib.lines.Line2D at 0x7f9b2f0df160>,
 <matplotlib.lines.Line2D at 0x7f9b29063ac8>,
 <matplotlib.lines.Line2D at 0x7f9b29063d30>,
 <matplotlib.lines.Line2D at 0x7f9b29063f98>,
 <matplotlib.lines.Line2D at 0x7f9b28fef240>,
 <matplotlib.lines.Line2D at 0x7f9b28fef4a8>,
 <matplotlib.lines.Line2D at 0x7f9b28fef710>,
 <matplotlib.lines.Line2D at 0x7f9b28faf720>]

In : plt.legend(["TIC",] + data.ref_cpds) # This isn't necessary
Out: <matplotlib.legend.Legend at 0x7f9b25a35438>

In : plt.show()
```

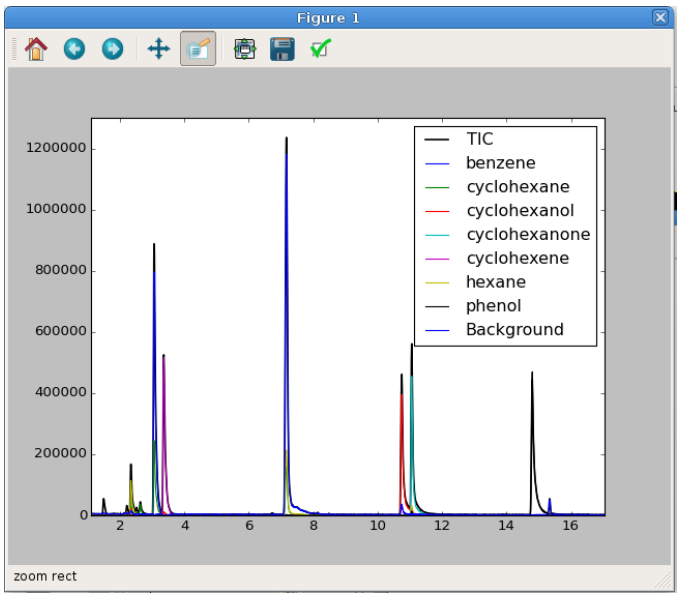

Figure 5.1: An interactive plot of the TIC and the NNLS simulated fits. This has been zoomed in to highlight the fit and data.

CHAPTER

SIX

DATA STORAGE

Processed data files can be stored on-disk using the `GcmsStore` object located in the `gcmstools.datastore` module. Not only does this create a convenient storage solution for processed data sets, it is also necessary when running calibrations on a group of related data sets. The file is a HDF file, which is an open-source high performance data storage container optimized for numerical data. Creation and manipulation of this file is controlled using a combination of two Python libraries: PyTables and Pandas. PyTables provides a high-level interface to create and modify HDF files, and Pandas is a very powerful package for working with tabular data. Both of these project have extensive documentation of their many advanced features, so little detail on their usage is provided here.

6.1 About GcmsStore Implementation

The `GcmsStore` object is a subclass of the Pandas `HDFStore` object, so it will contain all of the *functions described for that object*. The `GcmsStore` class simply adds a number of custom functions specific for GCMS data sets.

6.2 Create/Open the Container

A *gcmstools* `GcmsStore` object must be created with a file name argument. If a file with this name already exists, it will be opened for appending or modification. The default behavior is to compress all the data going into this file using the 'blosc' compression library and the highest compression level (9). See the Pandas `HDFStore` documentation for other accepted keyword arguments, especially the compression arguments if different values are required.

```
In : from gcmstools.datastore import GcmsStore

In : h5 = GcmsStore('data.h5')
```

6.3 Closing the File

In general, you will want to close the HDF file when you're done, although this is not strictly necessary.

```
In : h5.close() # Only do this when you're done
```

6.4 Recompressing the HDF File

HDF files are designed to be written once and read many times. If you are repeatedly adding new files to the HDF storage container, the file size may become much larger than seems necessary. You can recompress the file using the `compress` method (which first closes the HDF file).

```
In : h5.compress() # This closes the file as well.
```

6.5 Adding Data

Added files to this storage container is done using the `append_files` method, which can take either a single data object or a list of objects, if you have many objects to add at one time.

```
In : h5.append_files(data)
HDF Appending: datasample1.CDF

In : h5.append_files([otherdata1, otherdata2])
HDF Appending: otherdata1.CDF
HDF Appending: otherdata2.CDF
```

Data files can be added at any stage of the processing chain; however, the calibration process will not work properly if you don't reference/fit the data first. You can add an already existing data file as well. The GcmsStore object will check if that file is different than the saved version before overwriting the existing object. If it is not changed, then the file will be skipped.

6.6 Viewing the File List

You can see a list of the files that are stored in this file by viewing the `files` attribute, which is a Pandas DataFrame.

```
In : h5.files
          name         filename
0  datasample1  datasample1.CDF
1  otherdata1   otherdata1.CDF
2  otherdata2   otherdata2.CDF
```

There are two name columns in this table: "name" and "filename". The latter is the full file and path name as given when the GcmsFile object was created. Keep in mind that the path information may not be correct if you've moved the location of this storage file. In order to efficiently store the data on disk, the full file name is internally simplified the "name". This simplification removes the path and file extension from the file name. In addition, it replaces all ".", "-", and spaces characters with "_". If the file name starts with a number, the prefix "num" is added.

Warning: You will encounter problems if two or more file names simplify to the same "name". However, if you're file naming system does not produce unique file names for different data sets, you will most certainly have more problems in the long run than just using these programs.

6.7 Extracting Stored Data

You can extract data from the storage file using the `extract_gcms` method. This function takes one argument which is the name of the dataset that you want to extract. This name can be either the simplified name or the full filename (with or without the path). The extracted data is the same file object type as you stored originally.

```
In : extracted = h5.extract_gcms('datasample1')

In : extracted.filetype
Out: "AiaFile"
```

6.8 Stored Data Tables

This HDF data file may contain a number of Pandas data tables (DataFrames) with information about the files, calibration, etc. A list of currently available tables can be obtained by directly examining the `GcmsStore` by directly examining the `GcmsStore` instance. (Note: you won't see these attributes using *tab completion*.)

```
In : h5
Out:
<class 'pandas.io.pytables.HDFStore'>
File path: data.h5
/calibration            frame        (shape->[6,8])
/calinput               frame        (shape->[30,9])
/datacal                frame        (shape->[49,6])
/files                  frame        (shape->[1,2])
```

Directly viewing these tables is trivial.

```
In : h5.calibration
Out:
               Start  Stop  Standard         slope      intercept         r  \
Compound
benzene          2.9   3.5       NaN  38629.931565 -367129.586850  0.998767
phenol          14.6  15.1       NaN  30248.192619   65329.897933  0.999136
...

                     p       stderr
Compound
benzene       0.000052  1108.344872
phenol        0.000030   726.257380
...
```

More information on using these tables is provided in *Appendix B*.

CHAPTER

SEVEN

CALIBRATION AND CONCENTRATION DETERMINATION

7.1 Calibration Object

The calibration class `Calibrate` is defined in the `gcmstools.calibration` module. Before you start, you must have access to an HDF storage file that contains *all* of the data to be processed. By default, the `Calibrate` instance creation attempts to open a file "data.h5" in the current folder. If this is not the name of your HDF file, an alternate name can also be passed in on construction to open a different HDF file.

```
In : from gcmstools.calibration import Calibrate

In : cal = Calibrate() # Opens 'data.h5' by default

# Equivalent to above, but using a different file name, don't do both
In : cal = Calibrate('other.h5')
```

7.1.1 Closing the HDF File

In general, you will want to close the HDF file when you're done. This is not necessary, but it does ensure that the file gets properly compressed, which saves some disk space. If you don't do this, though, it won't hurt anything.

```
In : cal.close() # Only do this when you're done
```

7.2 Calibration Information File

In order to calibrate your GCMS data, you must first create a csv file containing all of the relevant calibration information. Again, the structure of this file is very important, so an example, "calibration.csv", is contained with the sample data.

```
In : from gcmstools.general import get_sample_data

In : get_sample_data("calibration.csv")
```

The first row in this csv file is critical, and it must look like this:

```
Compound,File,Concentration,Standard,Standard Conc
```

Each row after this describes a set of calibration information that you'd like to use. The columns of this file are as follows:

- *Compound*: The name of the compound that you are calibrating. This *must* correspond to one of the compound names (case-sensitive) used when *referencing and fitting* the GCMS file.

- *File*: This is the name of the data set that was collected at a particular concentration of *Compound*. Again, this filename can be the full filename (with or without the path) or the simplified name. See the *files attribute* section of the GcmsStore docs for more info.
- *Concentration*: This is the known concentration of the *Compound*. This should only be a number: do not include units. All of the concentrations should be in the same units, and keep in mind that all calibration and concentration data will then be in that same unit of measurement.
- *Standard* and *Standard Conc*: If there is an internal standard used in this file, you should provide the name and concentration in these columns. Again, the standard name should have been defined when referencing your GCMS data set, and do not include units with the concentration. Make sure your concentration units are the same as the reference compound to avoid confusion.

You can add extra columns to this table without penalty, in case you need to add additional information to this table. You can also comment out lines by starting a line with a `#` character. This is useful if you want to ignore a bad data point without completely removing the line from the calibration file.

7.3 Generate Calibration Curves

The calibration curves can be generated using the `curvegen` method of the `Calibrate` object. This function must be called with the name of your calibration file. In this example, that filename is "calibration.csv".

```
In : cal.curvegen('calibration.csv')
Calibrating: benzene
Calibrating: phenol
...

In :
```

This process creates two new tables as attributes to your calibration object, `calinput` and `calibration`. The former table is simply your input csv information with columns appended for the concentrations ("conc") and integrals ("integral") used for generating the calibration curve. If no internal standard is defined, then "conc" will be the same as the compound concentration you used in the input file. If an internal standard was defined, then "conc" and "integral" will be these values divided by the corresponding internal standard values. These tables also stored in the HDF file as well, if you want to check them at a later date.

The `calibration` table contains all of the newly created calibration curve information, such as slope, intercept, r value, etc.

```
In : cal.calibration
Out:
               Start  Stop  Standard         slope      intercept         r  \
Compound
benzene          2.9   3.5       NaN  38629.931565 -367129.586850  0.998767
phenol          14.6  15.1       NaN  30248.192619   65329.897933  0.999136
...

                     p       stderr
Compound
benzene       0.000052  1108.344872
phenol        0.000030   726.257380
...
```

7.3.1 Plotting Calibrations

By default, no plots are generated for these calibrations. There are, however, a couple of functions that automatically plot some of the calibration data.

1. `cal.curvegen('calibration.csv', calfolder='cal', picts=True)` : This invocation will auto generate pictures for all of the calibration compounds and place them in a folder defined by the keyword argument `calfolder`. This argument is optional, if you don't mind the default folder name of "cal". Be careful! This folder and its contents will be deleted before generating new plots, so if this folder exists, make sure it is clear of important data.
2. `cal.curveplot('benzene')` : This method will generate a plot of the benzene calibration information and save it to the current folder. There are several keyword arguments to this function:
   - `folder='.'` : This sets the folder where the picture will be saved. By default it is the current directory.
   - `show=False` : Change this value to `True` if you want an interactive plot window to be displayed. Default is `False`.
   - `save=True` : Save the calibration plot to the folder.

   If both `save` and `show` are set to `False`, nothing will happen.

   Of course, this function must be done after a call to `curvegen`. But it can be used to look at calibration data from an previously processed HDF file without rerunning the calibration.

7.4 Determine Sample Concentrations

Generating calibration curves *does not* automatically process the other data files. In order to determine concentrations for all of the remaining data in the HDF file, use the `datagen` method of the `Calibrate` object.

```
In : cal.datagen()
Processing: datasample1.CDF
Processing: otherdata1.CDF
Processing: otherdata2.CDF
...
```

After processing, another data table attribute (`datacal`) is created and saved to the HDF file.

```
In : cal.datacal
Out:
                              benzene       phenol   ...
name
datasample1               4239.070627    58.336917   ...
otherdata1                5475.778519    20.401981   ...
otherdata2                4355.094930    19.171877   ...
...
```

Note: Again, the data *ARE NOT* automatically integrated after generating calibration curves. If you change your calibration information by re-runing `curvegen`, you must re-run `datagen` to apply these changes to the other data sets contained in the HDF file.

7.4.1 Plotting Integrals with Concentrations

By default, no plots are generated for the integrals. If you'd like to see plots of the integrals, there are a couple of methods.

1. `cal.datagen(datafolder='data', picts=True)` : This method will auto generate pictures for all of the calibration compounds and place them in a folder defined by the keyword argument `datafolder`. This argument is optional, if you don't mind the default folder name of "data". Be careful! This will delete this folder before generating new plots, so if this folder exists, make sure it is clear of important data.
2. `cal.dataplot('benzene', 'datasample1')` : This method will generate a plot of the benzene integral for 'datasample1' and save it to the current folder. There are several keyword arguments to this function:
   - `folder='.'` : This sets the folder where the picture will be saved. By default it is the current directory.
   - `show=False` : Change this value to `True` if you want an interactive plot window to be displayed. Default is `False`.
   - `save=True` : Save the calibration plot to the folder.

   If both `save` and `show` are set to `False`, nothing will happen.

   Of course, this function call can only be done after a call to `datagen`, but it can be used to look at calibration data from an previously processed HDF file without rerunning the calibration and data integration functions.

CHAPTER

EIGHT

ISOTOPIC DISTRIBUTION ANALYSIS

When running reactions with isotopically labeled substrates, it may be important to determine the distribution of the isotope labels in the resulting product. The `gcmstools.isotope` module defines an `Isotope` class that is meant to assist in this analysis. In short, this class helps you to build a matrix of reference spectra which is used in a least squares fit with the observed mass distribution.

TODO

CHAPTER

NINE

BATCH PROCESSING

All of the previous steps can be automated using the `proc_data` fuction located in the `gcmstools.general` module.

```
In : from gcmstools.general import proc_data
```

This function only has two required arguments: 1) the path to the folder that contains *all* of the GCMS files and 2) the name of the HDF data file that you'd like to generate. In this example, our data is in the folder "data" and our processed data file is called "data.h5".

```
In : proc_data('data/', 'data.h5')
... # Lots of stuff will get printed at this point.
```

The `proc_data` function accepts numerous keyword arguments to allow some process control.

- *filetype='aia'* : This flag can be used to control the type of GCMS objects used for processing the data in your folder. The default value `'aia'` uses the `AiaFile` object. See *Basic Setup* for detailed information.
- *reffile=None* : Pass the name of a reference file for referencing your GCMS data. The default is `None`, so no referencing will be done. Otherwise, the reference object will be determined by the file extension. For example, `reffile='ref_specs.txt'` will create a `TxtReference` object using the file "ref_specs.txt" (which must exist of course). See *Referencing and Fitting* for more information.
- *fittype=None* : Set the fitting type to use for fitting the GCMS data. See *Referencing and Fitting* for detailed information. The valid choices are:
  - `'nnls'` for non-negative least squares fitting.
- *calfile=None* : Pass in the name of a calibration csv file to generate calibration curves and integrate the data. For example, `calfile='calibration.csv'` will calibrate your data using the information in the csv file "calibration.csv". See *Calibration and Concentration Determination* for detailed information, especially on the expected structure of the csv file.
- *picts=False* : Set this argument to true if you want to generate pictures of your calibration curves and data fits. These calibration curves will be placed in the folder "cal", and plots of integrals with concentrations will be place in the folder "proc". Be careful! All files in these folders will be deleted before the plots are generated. Also, if you have a lot of data files, this process can be very slow. It is possible to view these plots after processing. See *Calibration and Concentration Determination* for detailed information.
- *chunk_size=4* : This sets the number of files that will be processed at any given time. This keeps the total number of opened GCMS files to a minimum, which is important if you have a large number of files to process. You probably don't need to change this.
- *multiproc=False* : Setting this argument to `True` will use IPython's parallel machinery to run a lot of the processing using multiple cores on your processor. Before using this command, you must start an IPython node cluster from the command line. Open a new terminal, and type the following command:

```
home>$ ipcluster start -n 2
```

This will start a cluster of two (`-n 2`) nodes. You can have up to one node per core on your processor. Setting this number greater than the number of cores will result in a degradation of performance. To stop the node cluster, type `Ctrl-c` from the terminal where you started the cluster. Or else, you can stop it from another terminal using the following command:

```
home>$ ipcluster stop
```

This only works for processing the files, not for generating plots. So plotting your calibration data will still be very slow for a lot of data files.

See IPython's parallel documentation for more information.

- This function can also accept all keyword arguments for any file type, reference, fitting, and calibration objects. See their documentation for more information.

9.1 Complete Processing Command

A complete version of this processing command might look like the following.

```
In : proc_data('data/', 'data.h5', filetype='aia', reffile='ref_spects.txt',
        fittype='nnls', calfile='calibration.csv', multiproc=True)
### Lot's of stuff gets printed
...
```

The HDF file "data.h5" contains all of your data. See the *Data Storage*, *Calibration and Concentration Determination*, and *Appendix B* for more information on how to view and plot these data.

CHAPTER

TEN

APPENDIX A: RUNNING CODE SAMPLES

10.1 Using the Command Line

Running the code samples in this documentation requires a rudimentary knowledge of the command-line terminal (command prompt on Windows). The terminal can seem very "texty" and confusing at first; however, with a little practice it gets to be fairly intuitive and *efficient*. There are many tutorials online, for example, The Command Line Crash Course. However, a few basic command line concepts are covered here, for reference.

When you start a terminal, you will be presented with a window to type commands. In this documentation, the terminal command prompt will be denoted as `home>$`, where "home" indicates the current folder where the commands will be executed and ">$" is just a separator. These things do not need to be typed when entering commands. A similar format is common in a lot of online documentation. Some commands generate output. The output will occur after the command prompt, but will not be preceded by a command prompt symbol.

The most important thing you'll want to be able to do is move to different directories (i.e. folders). To do this there are a couple of useful terminal commands: "change directory" (`cd`) and "present working directory" (`pwd`). When you open the terminal, you will usually start in your "home" directory. This will probably be "/Users/username" on Mac, "/home/username" on Linux, or "C:\Users\username" on Windows. To move to a different directory, use the `cd` command; to see the location of the current folder, use `pwd`. Here's an example.

```
home$> pwd
/home/username/

home$> cd folder1

folder1$> pwd
/home/username/folder1
```

The second command here moved active directory to the folder called *folder1*. There are also a few special directory shortcuts:

- `~` This refers to the home directory.
- `..` (Double dot) This refers to the parent directory of the current directory.
- `.` (Single dot) This refers to the current directory.
- `\` or `/` Separators to combine directory names. The first works on Linux/Mac (and in IPython, see below), the second is required on Windows.

Here's these shortcuts in action.

```
home>$ cd folder1/folder2

folder2>$ pwd
/home/username/folder1/folder2
```

```
folder2>$ cd .

folder2>$ pwd
/home/username/folder1/folder2

folder2>$ cd ../..

home>$ pwd
/home/username

home>$ cd folder1/folder2

folder2>$ cd ~

home>$ pwd
/home/username

home>$ cd folder1/folder2

folder2>$ cd ~/folder3

folder3>$ pwd
/home/username/folder3
```

The other important command is `ls`, which lists the contents of the current directory. (In Windows, the equivalent command is `dir`.)

```
folder3>$ ls
file1 file2 folder4
```

10.2 IPython

10.2.1 Start IPython

One of Python's strengths as a data analysis language is its interactive interpreter. This mode is accessed from a terminal by typing `python`.

```
home>$ python
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
[GCC 4.7.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Any command that is typed at the >>> prompt is treated as Python code, and executed appropriately. That means you can interactively write and explore code in this manner.

```
>>> 2 + 2
4
>>> print('Hello World')
Hello World
```

The default Python interpreter is very limited, which is why IPython was developed. IPython is an advanced Python interpreter, which has several advanced features like autocompletion and introspection, just to name two. Over the years, this project has grown substantially, and in addition to a terminal based interpreter, there is now a GUI version and a very cool web-based Notebook as well. To learn more about the other features, consult the IPython documentation.

IPython is started from the terminal using the `ipython` command:

```
home>$ ipython
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: 2 + 2
Out[1]: 4

In [2]: print('Hello World')
Hello World
```

The `In [#]:` prompt now takes the place of >>> in the regular Python interpreter. In addition, certain types of output are preceded by an `Out[#]:` prompt. The numbers in brackets help you to determine the order that commands are processed. For this documentation, though, the numbers will be stripped for clarity, e.g. `In :` and `Out:`. If you see these prompts, you should know that the commands are being run in an IPython session.

10.2.2 Autocompletion and Introscpection

The take home message of this section is *use the Tab key a lot!* It will make you much more productive.

Two very nice aspect of the IPython interpreter are autocompletion and object introspection. Both of these will make use of the Tab key on your keyboard; in code snippets, this key will be denoted as `<tab>`, which means you should press the Tab key rather than typing it out. To see these two operations in action, we can first create a new string object.

```
In : my_string = 'Hello World'

In : print(my_string)
Hello World
```

To determine the methods available to a string object, we can use IPython's object introspection.

```
In : my_string.<tab>
my_string.capitalize    my_string.isidentifier  my_string.rindex
my_string.casefold      my_string.islower       my_string.rjust
my_string.center        my_string.isnumeric     my_string.rpartition
my_string.count         my_string.isprintable   my_string.rsplit
my_string.encode        my_string.isspace       my_string.rstrip
my_string.endswith      my_string.istitle       my_string.split
my_string.expandtabs    my_string.isupper       my_string.splitlines
my_string.find          my_string.join          my_string.startswith
my_string.format        my_string.ljust         my_string.strip
my_string.format_map    my_string.lower         my_string.swapcase
my_string.index         my_string.lstrip        my_string.title
my_string.isalnum       my_string.maketrans     my_string.translate
my_string.isalpha       my_string.partition     my_string.upper
my_string.isdecimal     my_string.replace       my_string.zfill
my_string.isdigit       my_string.rfind
```

As you can see, there are many, many things that you can do with this string object. IPython can also use the Tab key to autocomplete long names for variables, path strings, etc. Here's an example:

```
In : my_string.is<tab>
my_string.isalnum       my_string.isidentifier  my_string.isspace
my_string.isalpha       my_string.islower       my_string.istitle
my_string.isdecimal     my_string.isnumeric     my_string.isupper
my_string.isdigit       my_string.isprintable

In : my_string.isi<tab>
```

Notice that when you type tab here IPython automatically expands this to `my_string.isidentifier`. This works for path strings as well.

Note: It should be pointed out that tab completion also works on the regular command line terminal interface as well.

10.2.3 Magic Commands

IPython has a number of special, non-Python, commands that make its interpreter behave much like a command-line terminal. These commands, called Magic Commands, are preceded by `%` or `%%`. The magic command documentation covers many of these, but a few that are useful to the examples in this document are discussed here.

The magics `%cd`, `%pwd`, and `%ls` serve the exact same purpose as in the terminal. Another very useful magic is `%run`. This command executes a Python program file from inside the IPython session, and in addition to executing the code, it also loads the data and variables into the current IPython session. This is best explained by example. Create a new folder called `folder1` in your home directory. Create the file `test.py` in `folder1` and paste the following code into that file. (See *Working with Text Files* for some information on text files and Python programs.)

```
var1 = 7
var2 = "Hello World"
var3 = var1*var2
```

Now let's start up IPython and run this new program.

```
home>$ ipython
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In : %pwd
/home/username

In : %cd folder1
/home/username/folder1

In : %ls
test.py

In : %run test.py

In :
```

At this point, it seems like nothing has happened; however, the variable that we defined in our file "test.py" are now contained in our IPython session. Assuming that the following IPython code is the same session as above.

```
In : var1
Out: 7

In : var3
Out: Hello WorldHello WorldHello WorldHello WorldHello WorldHello
WorldHello WorldHello World
```

As you can see, this is a very powerful way to save your work for later or to run code that is fairly repetitive.

10.2.4 Notebook Interface

Todo.

10.3 Working with Text Files

There are many instances where you will need to work with plain text files, including writing Python programs. Plain text files are *not* word processing documents (e.g. MS Word), so you will want to use a dedicated text editor. Another source of problems for beginners is that leading white space in Python programs is important. For these reasons, a dedicated Python text editor can be very useful for beginners. Anaconda is bundled with Spyder, which has a builtin text editor. The Anaconda FAQ has information on running Spyder on your system. Spyder is actually a full development environment, so it can be very intimidating for beginners. Don't worry! The far left panel is the text editor, and you can use that without knowing what any of the other panels are doing. Some internet searches will reveal other text editors if you'd prefer something smaller. (Do *not* use MS Notepad.)

The ".py" suffix for Python programs can be important. On Windows, however, file extensions are not shown by default, which makes them difficult to modify. In these cases, you may inadvertently create a file with the extension ".py.txt", which will not behave as you expect. Consult the internet for ways to show file extensions on a Windows machine.

CHAPTER

ELEVEN

APPENDIX B

This appendix contains some examples of common usage scenarios. In some cases, the code will be presented as it would be written in a Python program file. These can be run from the command line as follows:

```
datafolder>$ python program_file.py
```

In addition, these scripts can be run from the IPython interpreter by using `%run` or by individually typing each line. See the *IPython instructions* for more information. Little explanation is provided here. See the documentation for detailed descriptions of the *gcmstools* objects or any third-party packages.

11.1 Full Manual Processing Example

Below is an example script that does a complete analysis starting from loading the files all the way through running a calibration and extracting concentrations. This is approximately equivalent to what the function `data_proc` is doing (see *Batch Processing* for a description that function).

This program is saved in the sample data folder under the name "full_manual.py".

```
import os

from gcmstools.filetypes import AiaFile
from gcmstools.reference import TxtReference
from gcmstools.fitting import Nnls
from gcmstools.datastore import GcmsStore
from gcmstools.calibration import Calibrate

datafolder = 'data/'

files = os.listdir(datafolder)
cdfs = [os.path.join(datafolder, f) for f in files if f.endswith('CDF')]

ref = TxtReference('ref_specs.txt')
fit = Nnls()
h5 = GcmsStore('data.h5')

for cdf in cdfs:
    data = AiaFile(cdf)
    ref(data)
    fit(data)
    h5.append_gcms(data)
h5.close()

cal = Calibrate()
cal.curvegen('calibration.csv')
```

```
27 cal.datagen()
28 cal.close()
```

A couple of notes. `os.listdir` is simply returning a list of all the files that are contained in the folder defined by `datafolder` ("data/").

Because that folder may have many different types of files, we need to do some filtering. `cdfs` uses Python's list comprehension to build up a list of filenames that end with "CDF". It also appends the `datafolder` path to the beginning of the file name. This list comprehension syntax is a very compact and efficient way to represent the following loop structure:

```
# The following for loop is equivalent to
# cdfs = [os.path.join(datafolder, f) for f in files if f.endswith('CDF')]
cdfs = []
for f in files:
    if f.endswith('CDF'):
        path_plus_name = os.path.join(datafolder, f)
        cdfs.append(path_plus_name)
```

11.2 Extracting Calibration Tables to Excel

It may be cumbersome to view calibration and concentration data directly from the HDF file. You can save any of the tables in the HDF file to an Excel format using Pandas' to_excel function which is part of each DataFrame.

```
from gcmstools.datastore import GcmsStore

h5 = GcmsStore('data.h5')
h5.datacal.to_excel('datacal.xlsx',)
h5.close()
```

Keep in mind that these Excel files are not tied in any way to the original HDF file. If you change the calibration information or add new data, you'll have to regenerate this file to see the new data.

11.3 Plotting a Mass Spectrum

You may want to be able to plot a mass spectrum from a GCMS file. This is fairly straightforward, but requires a few steps.

Step 1

First read in your file. This can be done directly:

```
In : from gcmstools.filetypes import AiaFile

In : data = AiaFile('datasample1.CDF')
Building: datasample1.CDF'
```

Or by reading out a data file from a HDF storage file:

```
In : from gcmstools.datastore import GcmsStore

In : h5 = GcmsStore('data.h5')

In : data = h5.extract_gcms('datasample1')
```

Step 2: Optional

Plot the total ion chromatogram to find the elution time for the peak of interest. This is only necessary if you don't already know the time.

```
In : import matplotlib.pyplot as plt

In : plt.plot(data.times, data.tic)
Out :
[<matplotlib.lines.Line2D at 0x7f34>]

In: plt.show()
```

Hover the mouse over the peak you want. Somewhere at the bottom on the window, it should display the x and y coordinates of the cursor. Note the x number, and close the plot window. You don't need to know the number to every last digit of precision – a rough approximation will probably get you close.

Step 3

Now you can find the array index that corresponds to that time. All GCMS file objects define an `index` function to help you with this. The first argument of the index function is the array to index. After that you can put one or more numbers. For a single input number, a single index will be returned. For several values, a list of indices will be returned. In "datasample1.CDF", octane elutes at approximately 7.16 minutes.

```
In : idx = data.index(data.times, 7.16)

In : idx
Out: 1311
```

Step 4

Now we are ready to plot the data. The `GmsFile.intensity` attribute is a 2D array of all the measured MS intensities at every time point. Its shape is (number of time points, number of masses). We can use our time index to select a certain time. We'll make this a box plot, which is more typical for this type of data. (You only need to import Matplotlib if you haven't done it before.)

```
In : import matplotlib.pyplot as plt

In : plt.bar(data.masses, data.intensity[idx])
Out: <Container object of 462 artists>

In : plt.show()
```

You should see something like the plot in Figure 11.1.

11.4 Plotting Data and Reference MS

Let's say you want to compare the reference MS with your data set. This can be done in much the same manner as before but with a couple of extra steps. We'll use 'benzene' as our reference compound of choice.

Step 1

Get your *referenced* MS data.

```
In : from gcmstools.filetypes import AiaFile

In : from gcmstools.reference import TxtReference

In : data = AiaFile('datasample1.CDF')
```

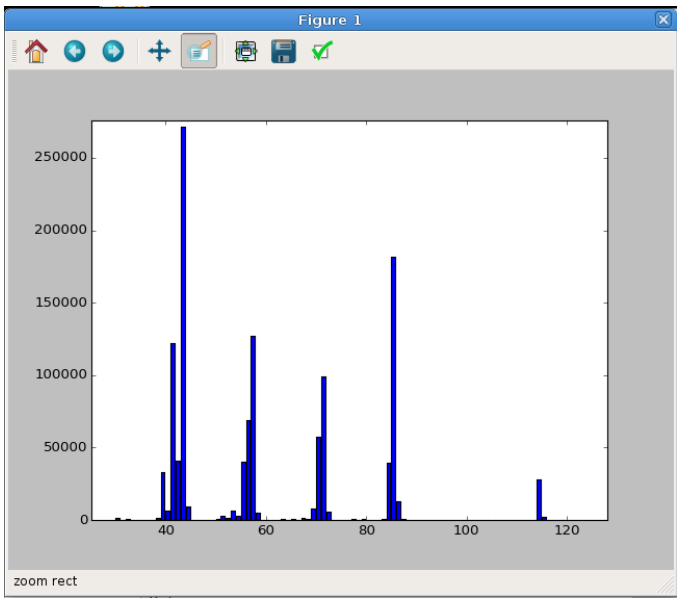

Figure 11.1: The MS of octane plotted from our sample data set. In this case, the plot was zoomed in a bit to highlight the relevant data region. Your initial plot may look different.

```
Building: datasample1.CDF'

In : ref = TxtReference('ref_specs.txt')

In : ref(data)
Referencing: datasample1.CDF
```

Or from the HDF file.

```
In : from gcmstools.datastore import GcmsStore

In : h5 = GcmsStore('data.h5')

In : data = h5.extract_gcms('datasample1.CDF')
```

Step 2

Get the time index as shown above. Benzene elutes at 3.07 minutes.

```
In : idx = data.index(data.times, 3.07)

In : idx
Out: 553
```

You'll also need an index for the reference compound.

```
In : refidx = data.ref_cpds.index('benzene')

In : refidx
Out: 0
```

Step 3

The reference MS are stored in the `ref_array` variable. This is also a 2D numerical array with the shape of (number of ref compounds, number of masses), so indices can be selected in the same manner as before. However, these data

are normalized, and the sample data will be much larger. We can easily normalize the data spectrum.

```
In : normdata = data.intensity[idx]/data.intensity[idx].max()
```

Import Matplotlib for plotting before doing anything else.

```
In : import matplotlib.pyplot as plt
```

11.4.1 Side-by-Side Plot

Here we'll do a side-by-side plot. By default, each bar will take up all of the space between each x-axis point (1.0 in this case), so we need to make the bars narrower otherwise the second bar plot will overlap. In addition, the x-axis for the second data set must be adjusted so the bars start at slightly adjusted positions. We'll also change the face color ("fc") otherwise they will be the same for both. In this case, we'll use the same adjustment as the width of the bars. The resulting plot is show in Figure 11.2.

```
In : plt.bar(data.masses, normdata, width=0.5, fc='b')
Out: <Container object of 462 artists>

In : plt.bar(data.masses+0.5, data.ref_array[refidx], width=0.5, fc='r')
Out: <Container object of 462 artists>

In : plt.show()
```

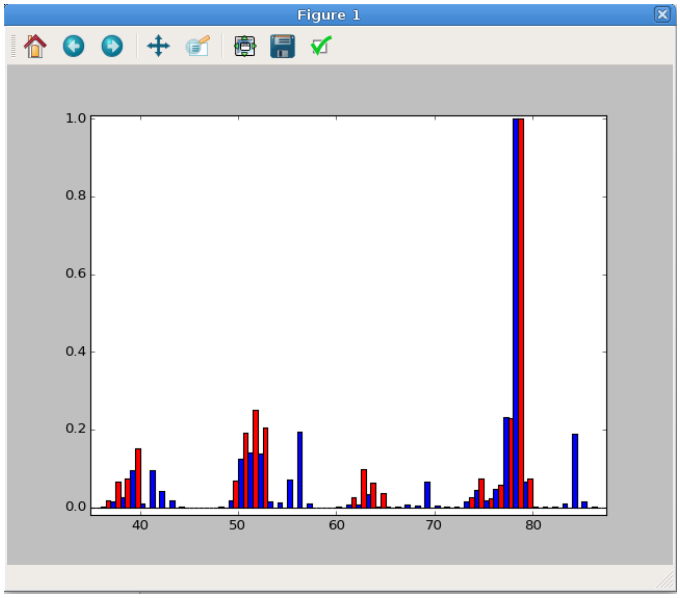

Figure 11.2: A side-by-side MS plot. This has been zoomed in to highlight the important data region.

11.4.2 Up-Down Plot

As an alternative, you could plot one of the data sets upside down, which may have some utility. Notice, we just need to invert (−) one of the intensity data sets. The resulting plot is shown in Figure 11.3.

```
In : plt.bar(data.masses, normdata, fc='b')
Out: <Container object of 462 artists>
```

```
In : plt.bar(data.masses+0.5, -data.ref_array[refidx], fc='r')
Out: <Container object of 462 artists>

In : plt.show()
```

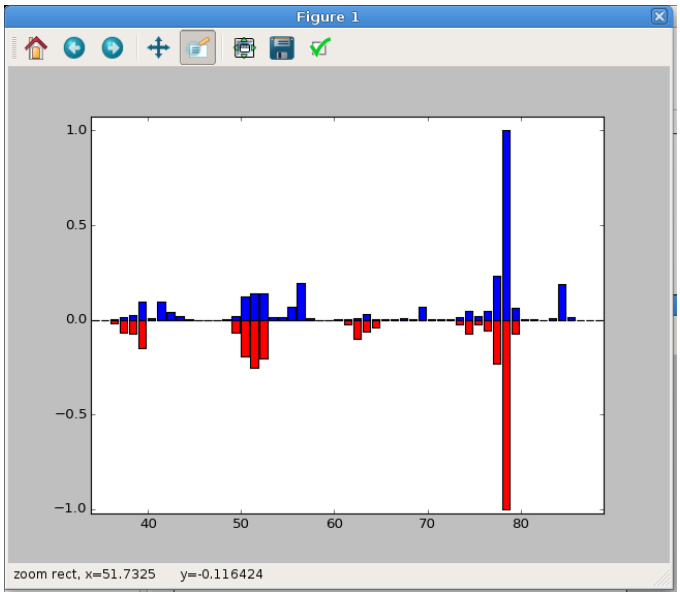

Figure 11.3: A Up-Down MS plot. This has been zoomed in to highlight the important data region.

11.4.3 Difference Plot

To plot the difference between the data and the reference, just subtract one spectrum form the other. The resulting plot is shown in Figure 11.4.

```
In : diff = normdata - data.ref_array[refidx]

In : plt.bar(data.masses, diff)
Out: <Container object of 462 artists>

In : plt.show()
```

11.5 Plotting Data and Fitted MS

You may want to plot a mass spectrum relative to the fitted spectrum for one particular reference compound. This can be done in a very similar manner as above, but we must first fit our data. This will follow the side-by-side reference plotting section, check there to find out how to set some of these variables. Again, we'll use 'benzene' as our test case.

```
In : from gcmstools.fitting import Nnls

In : fit = Nnls()

In : fit(data)
Fitting: datasample1.CDF
```

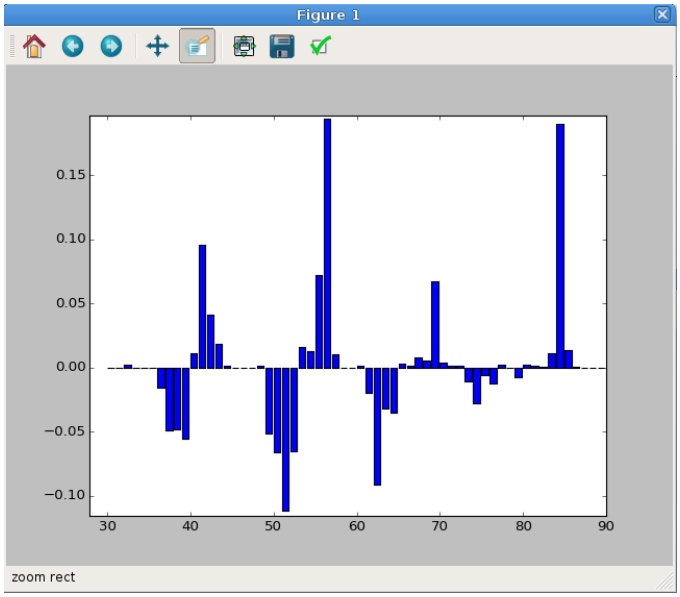

Figure 11.4: A difference mass spectrum plot. This has been zoomed in to highlight the important data region.

11.5.1 Bar Plot

In this case, we need to do a little Numpy index magic. Remember that the fitting generate a `fit_coef` array. These are the least-squares coefficients at every data point. This is a 2D array with shape (# of time points, # of reference compounds). We will use our time index (`idx`) and our reference index (`refidx`) to select out one least-squares coefficient and then multiply this by our reference mass spectrum. The final plot window is shown in Figure 11.5.

```
In : fitspec = data.fit_coef[idx, refidx]*data.ref_array[refidx]

In : plt.bar(data.masses, data.intensity[idx], width=0.5, fc='b')
Out: <Container object of 462 artists>

In : plt.bar(data.masses+0.5, fitspec, width=0.5, fc='r')
Out: <Container object of 462 artists>

In : plt.show()
```

11.5.2 Dot Plot

Alternatively, this comparison can be done with a standard plot, as long as the markers are set to be large dots (`'o'`). A perfect correlation would be a straight line. We can show this by plotting the sample data against itself as a solid black line (`'k-'`). This plot window is shown in Figure 11.6.

```
In : fitspec = data.fit_coef[idx, refidx]*data.ref_array[refidx]

In : plt.plot(data.intensity[idx], fitspec, 'o')
[<matplotlib.lines.Line2D at 0x7f34>]

In : plt.plot(data.intensity[idx], data.intensity[idx], 'k-')
[<matplotlib.lines.Line2D at 0x7f34>]

In : plt.show()
```

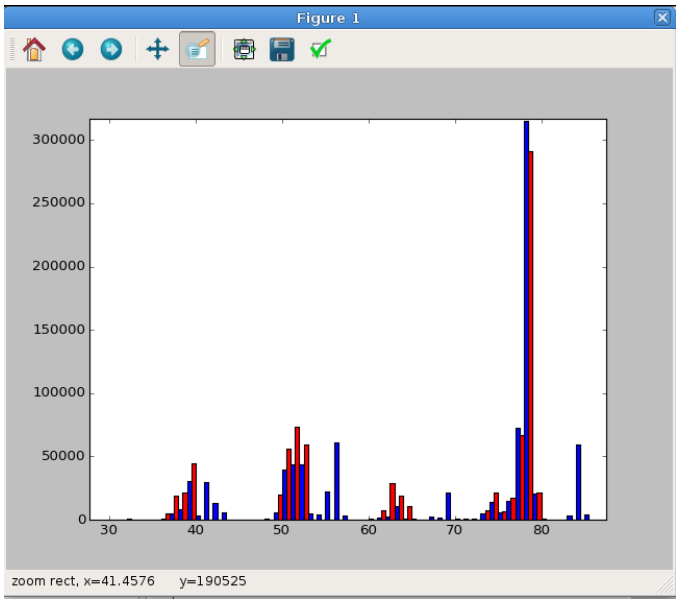

Figure 11.5: A side-by-side MS plot of the sample data versus the fitted data for benzene. This has been zoomed in to highlight the important data region.

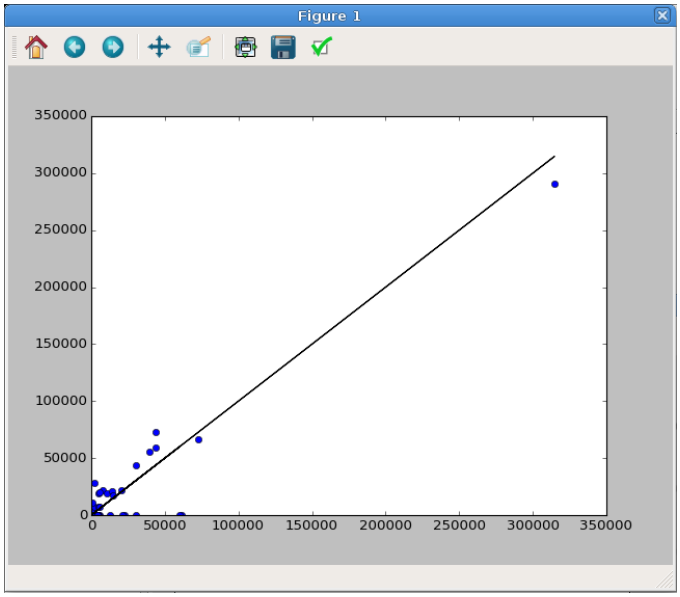

Figure 11.6: A dot plot comparing the sample data versus the fitted data for benzene. A perfectly straight line (black) indicates a perfect fit.

11.5.3 Fancy Data Versus Fit MS Plot

This example will be very similar to the Data vs Fit plots above; however, we will use a fully process HDF storage file to get our data set. This will be presented as a complete script. A copy of which, called "fancy_ms.py", is contained in the sample data folder along with a copy of the PDF output ("fancy_ms.pdf"). A picture of the resulting plot is shown in Figure 11.7.

```
import matplotlib.pyplot as plt

from gcmstools.datastore import GcmsStore

h5 = GcmsStore('data.h5')
data = h5.extract_gcms('data1')
h5.close()

refcpd = 'benzene'
refidx = data.ref_cpds.index(refcpd)

time = 3.07
idx = data.index(data.times, time)

fitspec = data.fit_coef[idx, refidx]*data.ref_array[refidx]
dataspec = data.intensity[idx]

### This is all the plotting stuff

plt.figure(figsize=(10,4))

plt.bar(data.masses, dataspec, width=0.5, fc='b', label="Sample")
plt.bar(data.masses+0.5, fitspec, width=0.5, fc='r', label="Fit")
plt.grid()

plt.xlim(30, 90)
plt.xlabel('m/z')
# Don't show the y-axis ticks
plt.tick_params(axis='y', labelleft=False)
plt.title('Benzene Mass Spectra')

plt.tight_layout()
# We can save files in PDF format, which gives them unlimited resolution.
plt.savefig('fancy_ms.pdf')
```

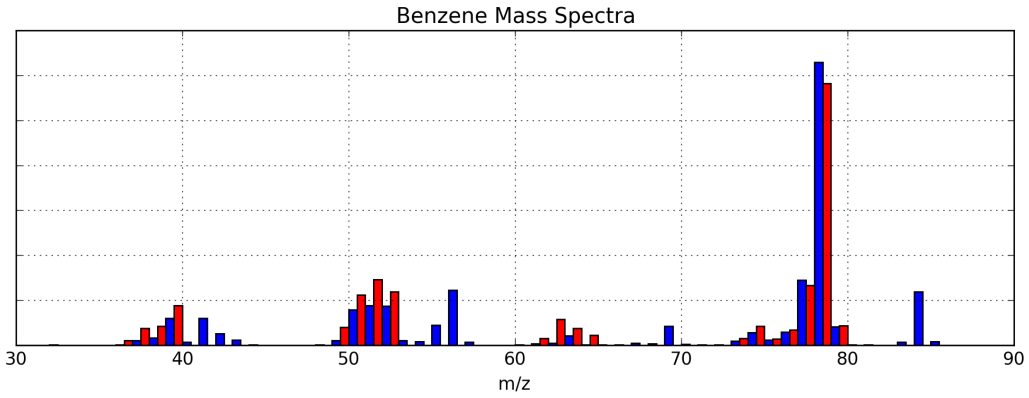

Figure 11.7: A fancy plot comparing the sample data (blue) and the fitted data (red) for benzene. The PDF version is a vector graphic, so it has unlimited resolution.