MCALF

Release 1.0.0

Conor D. MacBride & David B. Jess

Mar 28, 2023

CONTENTS:

CHAPTER

ONE

MCALF DOCUMENTATION

Welcome to MCALF's documentation!

MCALF is an open-source Python package for accurately constraining velocity information from spectral imaging observations using machine learning techniques.

These pages document how the package can be interacted with. Some examples are also provided. A `Documentation Index` and a `Module Index` are available.

1.1 User Documentation

License

MCALF is an open-source Python package for accurately constraining velocity information from spectral imaging observations using machine learning techniques.

This software package is intended to be used by solar physicists trying to extract line-of-sight (LOS) Doppler velocity information from spectral imaging observations (Stokes I measurements) of the Sun. A 'toolkit' is provided that can be used to define a spectral model optimised for a particular dataset.

This package is particularly suited for extracting velocity information from spectral imaging observations where the individual spectra can contain multiple spectral components. Such multiple components are typically present when active solar phenomenon occur within an isolated region of the solar disk. Spectra within such a region will often have a large emission component superimposed on top of the underlying absorption spectral profile from the quiescent solar atmosphere.

A sample model is provided for an IBIS Ca II 8542 Å spectral imaging sunspot dataset. This dataset typically contains spectra with multiple atmospheric components and this package supports the isolation of the individual components such that velocity information can be constrained for each component. Using this sample model, as well as the separate base (template) model it is built upon, a custom model can easily be built for a specific dataset.

The custom model can be designed to take into account the spectral shape of each particular spectrum in the dataset. By training a neural network classifier using a sample of spectra from the dataset labelled with their spectral shapes, the spectral shape of any spectrum in the dataset can be found. The fitting algorithm can then be adjusted for each spectrum based on the particular spectral shape the neural network assigned it.

This package is designed to run in parallel over large data cubes, as well as in serial. As each spectrum is processed in isolation, this package scales very well across many processor cores. Numerous functions are provided to plot the results in a clearly. The MCALF API also contains many useful functions which have the potential of being integrated into other Python packages.

1.1.1 Installation

For easier package management we recommend using Miniconda (or Anaconda) and creating a new conda environment to install MCALF inside. To install MCALF using Miniconda, run the following commands in your system's command prompt, or if you are using Windows, in the 'Anaconda Prompt':

```
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict
$ conda install mcalf
```

MCALF is updated to the latest version by running:

```
$ conda update mcalf
```

Alternatively, you can install MCALF using `pip`:

```
$ pip install mcalf
```

1.1.2 Testing

A test suite is included with the package. The package is tested on multiple platforms, however you may wish to run the tests on your system also.

Installing Dependencies

Using MCALF with pip

To run the tests you need a number of extra packages installed. If you installed MCALF using pip, you can run `pip install "mcalf[tests]"` to install the additional testing dependencies (and MCALF if it's not already installed).

Using MCALF with conda

If you want to use MCALF inside a conda environment you should first follow the conda installation instructions above. Once MCALF is installed in a conda environment, ask conda to install each of MCALF's testing dependencies using the following command. (See setup.cfg for an up-to-date list of dependencies.)

```
$ conda install pytest pytest-cov pytest-mpl tox
```

Running Tests

Tests should be run within the virtual environment where MCALF and its testing dependencies were installed. Run the following command to test your installation,

```
$ pytest --pyargs mcalf
```

Editing the Code

If you are planning on making changes to your local version of the code, it is recommended to run the test suite to help ensure that the changes do not introduce problems elsewhere.

Before making changes, you'll need to set up a development environment. The SunPy Community have compiled an excellent set of instructions and is available in their documentation. You can mostly replace `sunpy` with `mcalf`, and install with

```
$ pip install -e ".[tests,docs]"
```

After making changes to the MCALF source, run the MCALF test suite with the following command (while in the same directory as `setup.py`),

```
$ pytest --pyargs mcalf --cov
```

The tox package has also been configured to run the MCALF test suite.

1.1.3 Getting Started

The following examples provide the key details on how to use this package. For more details on how to use the particular classes and function, please consult the Code Reference.

Example Gallery

Here are a collection of examples on how this package can be used.

Models

Below are examples of how to use the models included within the models module:

Visualisation

Below are examples of plots produced using functions within the visualisation module:

Models

Below are examples of how to use the models included within the models module:

Working with IBIS data

This example shows how to initialise the *mcalf.models.IBIS8542Model* class with real IBIS data, and train a neural network classifier. We then proceed to fit the array of spectra and visualise the results.

Download sample data

First, the sample data needs to be downloaded from the GitHub repository where it is hosted. This will create four new files in the current directory (about 651 KB total). **You may need to install the requests Python package for this step to run.**

```
import requests

path = 'https://raw.githubusercontent.com/ConorMacBride/mcalf/main/examples/data/
↪ibis8542data/'

for file in ('wavelengths.txt', 'spectra.fits',
             'training_data.json', 'results.fits'):
    r = requests.get(path + file, allow_redirects=True)
    with open(file, 'wb') as f:
        f.write(r.content)
```

Load the sample data

Next, the downloaded data needs to be loaded into Python.

```
# Import the packages needed for loading data
import json
import numpy as np
from astropy.io import fits

# Load the spectra's wavelength points
wavelengths = np.loadtxt('wavelengths.txt', dtype='>f4')

# Load the array of spectra
with fits.open('spectra.fits') as hdul:
    spectra = hdul[0].data

# Load indices of labelled spectra
with open('training_data.json', 'r') as f:
    data = f.read()
training_data = json.loads(data)
```

As you can see, the sample data consists of a 60 by 50 array of spectra with 27 wavelength points,

```
print(wavelengths.shape, spectra.shape)
```

```
(27,) (27, 60, 50)
```

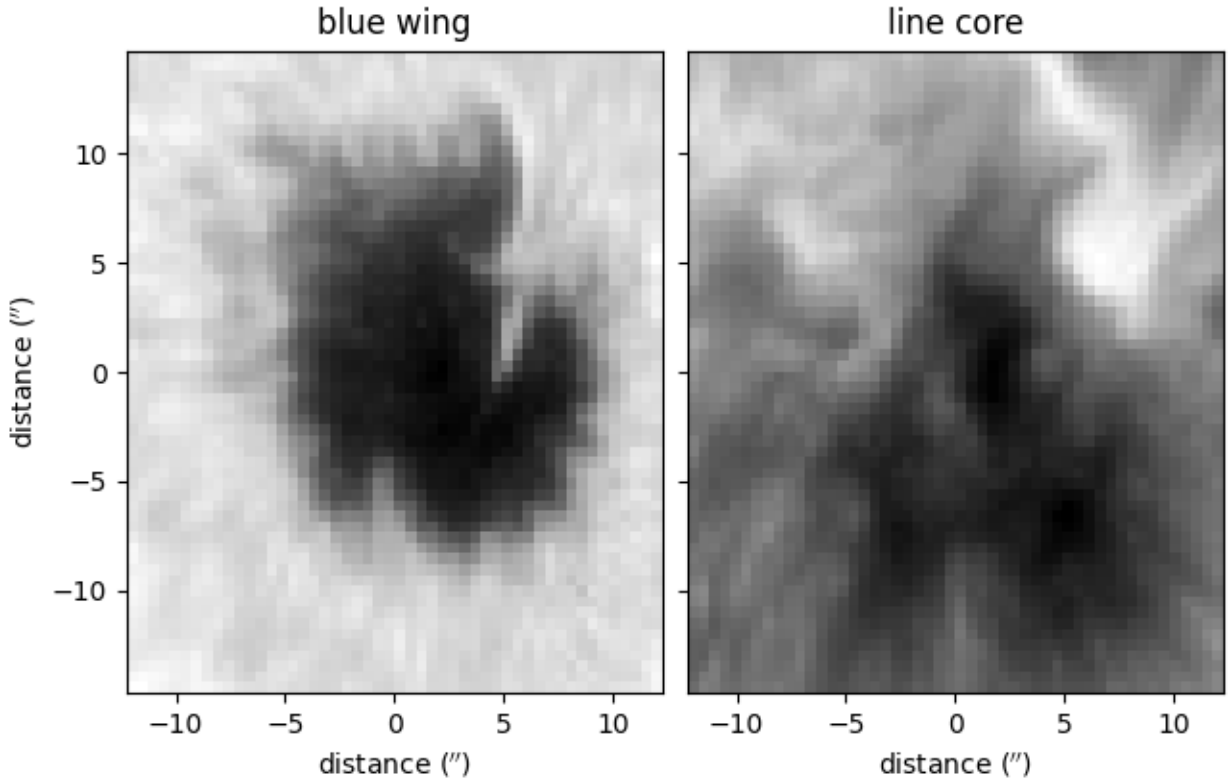
The blue wing and line core intensity values of the spectra are plotted below for illustrative purposes,

```
import matplotlib.pyplot as plt
import astropy.units as u
from mcalf.visualisation import plot_map

fig, ax = plt.subplots(1, 2, sharey=True, constrained_layout=True)
```

(continues on next page)

(continued from previous page)

```
wing_data = np.log(spectra[0])
core_data = np.log(spectra[len(wavelengths)//2])

res = {
    'offset': (-25, -30),
    'resolution': (0.098 * 5 * u.arcsec, 0.098 * 5 * u.arcsec),
    'show_colorbar': False,
}

wing = plot_map(wing_data, ax=ax[0], **res,
                vmin=np.min(wing_data), vmax=np.max(wing_data))
core = plot_map(core_data, ax=ax[1], **res,
                vmin=np.min(core_data), vmax=np.max(core_data))

wing.set_cmap('gray')
core.set_cmap('gray')

ax[0].set_title('blue wing')
ax[1].set_title('line core')
ax[1].set_ylabel('')

plt.show()
```



Generate the backgrounds array

As discussed in the *Using IBIS8542Model* example, a background intensity value must be specified for each spectrum.

For this small sample dataset, we shall simply use the average of the four leftmost intensity values of each spectrum,

```
backgrounds = np.mean(spectra[:4], axis=0)
```

Initialise the IBIS8542Model

The loaded data can now be passed into an *mcalf.models.IBIS8542Model* object.

```
import mcalf.models

model = mcalf.models.IBIS8542Model(original_wavelengths=wavelengths, random_state=0)

model.load_background(backgrounds, ['row', 'column'])
model.load_array(spectra, ['wavelength', 'row', 'column'])
```

Training the neural network

By default, the *mcalf.models.IBIS8542Model* object is loaded with an untrained neural network,

```
model.neural_network
```

The *mcalf.models.IBIS8542Model* class provides two methods to train and test the loaded neural network.

The `training_data.json` file contains a dictionary of indices for each classification 0 to 5. These indices correspond to randomly pre-chosen spectra in the `spectra.fits` file.

The training set consists of 200 spectra; 40 for each classification. This training set is for demonstration purposes only, generally it is not recommended to train with such a relatively high percentage of your data, as the risk of overfitting the neural network to this specific 60 by 50 array is increased.

To begin, we'll convert the list of indices into a list of spectra and corresponding classifications,

```
from mcalf.utils.spec import normalise_spectrum


def select_training_set(indices, model):
    for c in sorted([int(i) for i in indices.keys()]):
        i = indices[str(c)]
        spectra = np.array([normalise_spectrum(
            model.get_spectra(row=j, column=k)[0, 0, 0],
            model.constant_wavelengths, model.constant_wavelengths
        ) for j, k in i])
        try:
            _X = np.vstack((_X, spectra))
            _y = np.hstack((_y, [c] * len(spectra)))
        except NameError:
            _X = spectra
            _y = [c] * len(spectra)
    return _X, _y
```

(continues on next page)

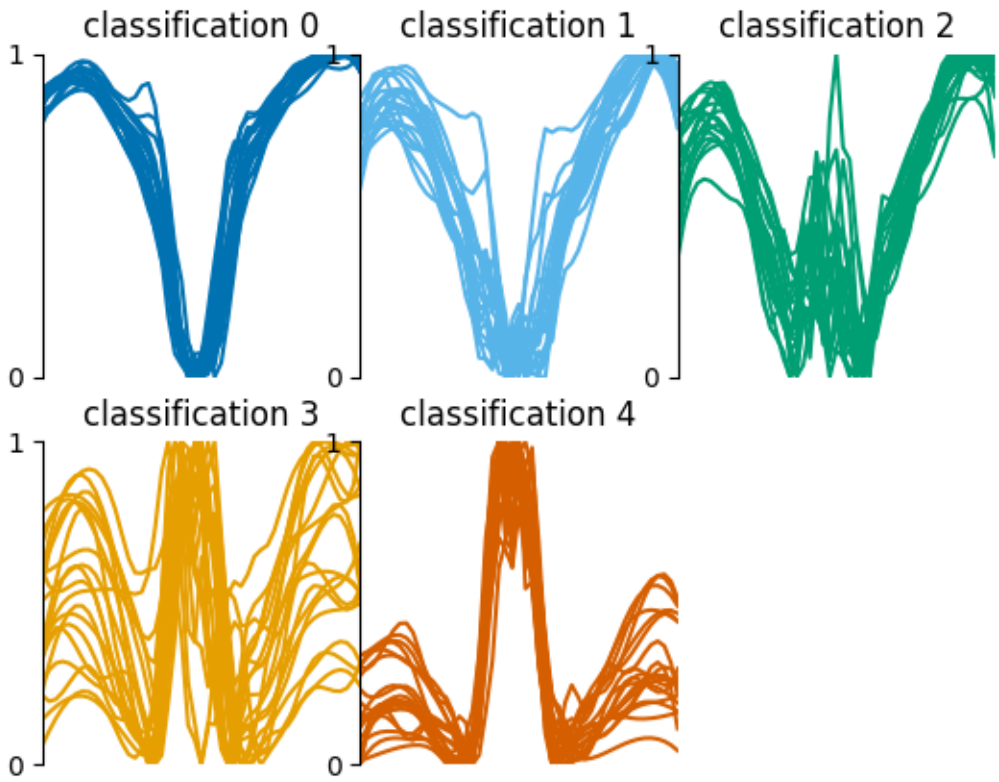
(continued from previous page)

```
X, y = select_training_set(training_data, model)

print(X.shape)  # spectra
print(y.shape)  # labels/classifications
```

```
(200, 49)
(200,)
```

These classifications look as follows,

```
from mcalf.visualisation import plot_classifications

plot_classifications(X, y)
```




```
GridSpec(2, 3)
```

Now we can train the neural network on 100 labelled spectra (even indices),

```
model.train(X[::2], y[::2])
```

And now we can use the other 100 labelled spectra (odd indices) to test the performance of the neural network,

```
model.test(X[1::2], y[1::2])
```

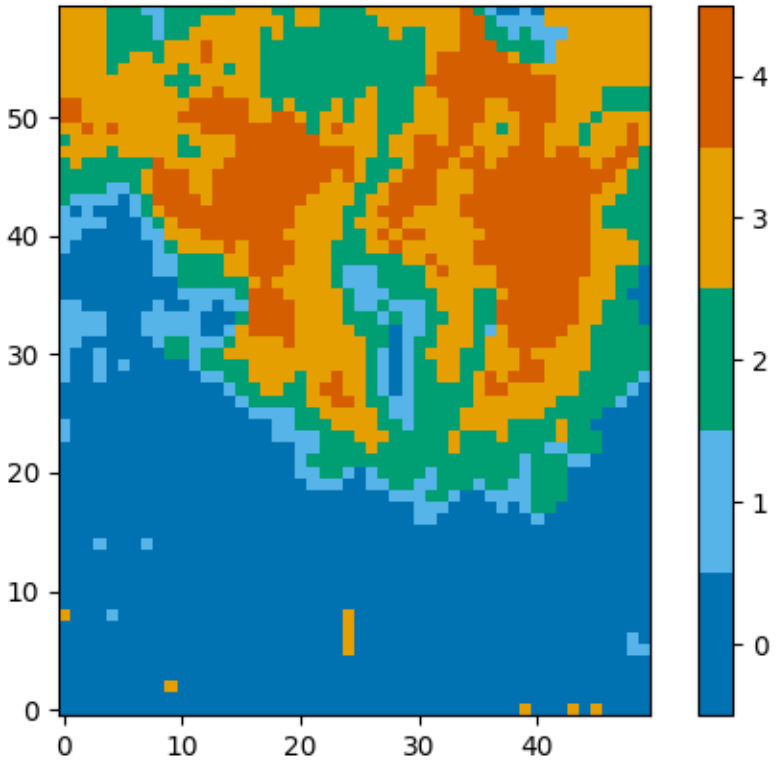
```
+---------------------------------------------+
|      Neural Network Testing Statistics      |
+---------------------------------------------+
| Percentage predictions==labels ::  89.00%   |
+---------------------------------------------+
| Average deviation for each classification   |
+---------------------------------------------+
|     class  0 ::   0.00 ±  0.00              |
+---------------------------------------------+
|     class  1 ::   0.20 ±  0.68              |
+---------------------------------------------+
|     class  2 ::   0.10 ±  0.44              |
+---------------------------------------------+
|     class  3 ::   0.00 ±  0.32              |
+---------------------------------------------+
|     class  4 ::  -0.05 ±  0.22              |
+---------------------------------------------+
| Average deviation overall ::   0.05 ±  0.41 |
+---------------------------------------------+
              precision    recall  f1-score   support

           0       0.95      1.00      0.98        20
           1       0.94      0.80      0.86        20
           2       0.89      0.80      0.84        20
           3       0.75      0.90      0.82        20
           4       0.95      0.95      0.95        20

    accuracy                           0.89       100
   macro avg       0.90      0.89      0.89       100
weighted avg       0.90      0.89      0.89       100
```

Now that we have a trained neural network, we can use it to classify spectra. Usually spectra will be classified automatically during the fitting process, however, you can request the classification by themselves,

```
classifications = model.classify_spectra(row=range(60), column=range(50))
```

These classifications look as follows,

```
from mcalf.visualisation import plot_class_map

plot_class_map(classifications)
```




```
<matplotlib.image.AxesImage object at 0x7fe9364fb040>
```

Creating a reproducible classifier

The neural network classifier introduces a certain amount of randomness when it it fitting based on the training data. This randomness arises in the initial values of the weights and biases that are fitted during the training process, as well as the order in which the training data are used.

This means that two neural networks trained on identical data will not produce the same results. To aid the reproducibility of results that rely on a neural network's classifications, a *random_state* integer can be passed to *mcalf.models.IBIS8542Model* as we did above. When we set this value to an integer, no matter how many times we train the neural network on the same data, it will always give the same results.

Until better solutions are available to store trained neural networks, a trained neural network can be saved to a Python pickle file and later reloaded. For maximum compatibility, it is recommended to reload into the same version of scikit-learn and its dependencies.

The neural network trained above can be saved as follows,

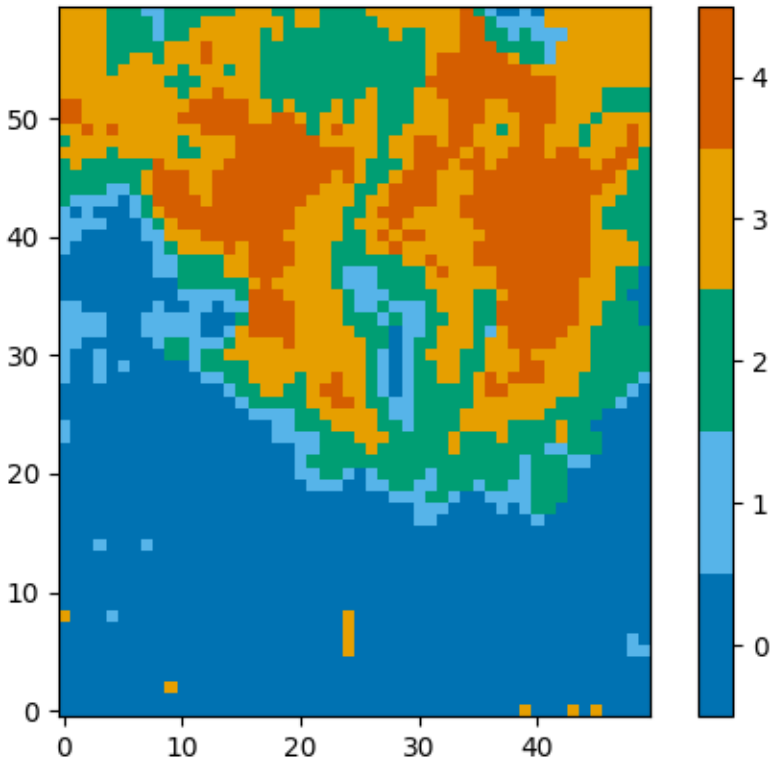
```
import pickle
pkl = open('trained_neural_network.pkl', 'wb')
pickle.dump(model.neural_network, pkl)
pkl.close()
```

This trained neural network can then be reloaded at a later date as follows,

```
import pickle
pkl = open('trained_neural_network.pkl', 'rb')
model.neural_network = pickle.load(pkl)  # Overwrite the default untrained model
```

And you can see that the classifications of spectra are the same,

```
plot_class_map(model.classify_spectra(row=range(60), column=range(50)))
```




```
<matplotlib.image.AxesImage object at 0x7fe9359d0af0>
```

Please see the scikit-learn documentation for more details on model persistence.

Fitting the spectra

Now that the data have been loaded and the neural network has been trained, we can proceed to fit the spectra.

Using pre-calculated results

As our 60 by 50 array contains 3000 spectra, it would take roughly 10 minutes to fit them all over 6 processing pools. We include pre-calculated results in the downloaded `results.fits` file.

The next step of the example loads this file back into Python as though we have just directly calculated it. This isn't something you would usually need to do, so do not worry about the contents of the `load_results()` function, however, we plan to include this functionality in MCALF itself in the future.

```
def load_results(file):

    with fits.open(file) as hdul:
        for hdu in hdul:
            if hdu.name == 'PARAMETERS':
                r_parameters = hdu.data.copy().reshape(-1, 8)
            elif hdu.name == 'CLASSIFICATIONS':
                r_classifications = hdu.data.copy().flatten()
                r_profile = np.full_like(r_classifications, 'both', dtype=object)
                r_profile[r_classifications <= 1] = 'absorption'
            elif hdu.name == 'SUCCESS':
                r_success = hdu.data.copy().flatten()
            elif hdu.name == 'CHI2':
                r_chi2 = hdu.data.copy().flatten()

    results = []
    for i in range(len(r_parameters)):
        fitted_parameters = r_parameters[i]
        fit_info = {
            'classification': r_classifications[i],
            'profile': r_profile[i],
            'success': r_success[i],
            'chi2': r_chi2[i],
            'index': [0, *np.unravel_index(i, (60, 50))],
        }
        if fit_info['profile'] == 'absorption':
            fitted_parameters = fitted_parameters[:4]
        results.append(mcalf.models.FitResult(fitted_parameters, fit_info))

    return results


result_list = load_results('results.fits')

result_list[:4]  # The first four
```

```
[Successful FitResult at (0, 0, 0) with absorption profile of classification 0,␣
↪Successful FitResult at (0, 0, 1) with absorption profile of classification 0,␣
↪Successful FitResult at (0, 0, 2) with absorption profile of classification 0,␣
↪Successful FitResult at (0, 0, 3) with absorption profile of classification 0]
```

Using your own results

You can run the following code to generate the `result_list` variable for yourself. Try starting with a smaller range of rows and columns, and set the number of pools based on the specification of your processor.

```
# result_list = model.fit(row=range(60), column=range(50), n_pools=6)
```

The order of the *mcalf.models.FitResult* objects in this list will also differ as the order that spectra finish fitting in each pool is unpredictable.

Merging the FitResult objects

The list of *mcalf.models.FitResult* objects can be mergerd into a *mcalf.models.FitResults* object, and then saved to a file, just like the `results.fits` file downloaded earlier.

First the object needs to be initialised with the spatial dimensions and number of fitted parameters,

```
results = mcalf.models.FitResults((60, 50), 8)
```

Now we can loop through the list of fits and append them to this object,

```
for fit in result_list:
    results.append(fit)
```

This object can then be saved to file,

```
results.save('ibis8542data.fits', model)
```

The file has the following structure,

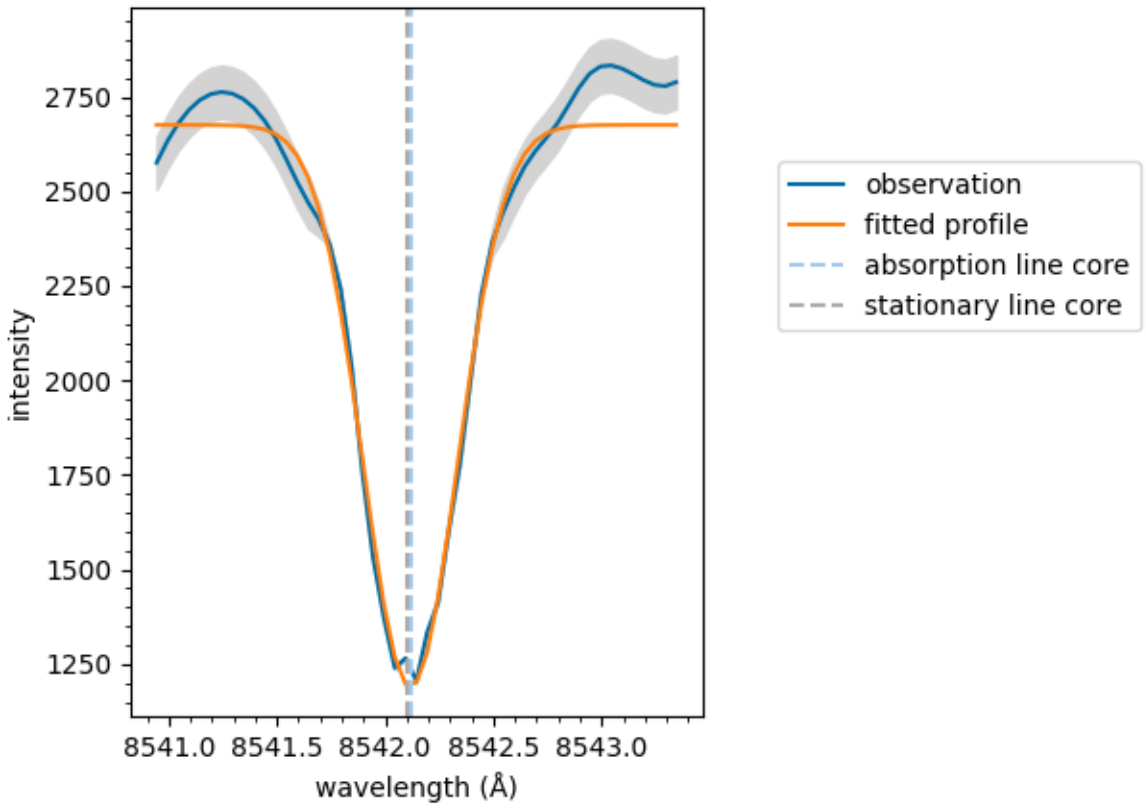
```
with fits.open('ibis8542data.fits') as hdul:
    hdul.info()
```

```
Filename: ibis8542data.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      13   (0,)
  1  PARAMETERS    1 ImageHDU        12   (8, 50, 60)   float64
  2  CLASSIFICATIONS    1 ImageHDU        10   (50, 60)   int16
  3  PROFILE       1 ImageHDU        11   (50, 60)   int16
  4  SUCCESS       1 ImageHDU        10   (50, 60)   int16
  5  CHI2          1 ImageHDU        10   (50, 60)   float64
  6  VLOSA         1 ImageHDU        12   (50, 60)   float64
  7  VLOSQ         1 ImageHDU        12   (50, 60)   float64
```

Exploring the fitted data

You can plot a fitted spectrum as follows,

```
model.plot(result_list[0])
```




```
<Axes: xlabel='wavelength (Å)', ylabel='intensity'>
```

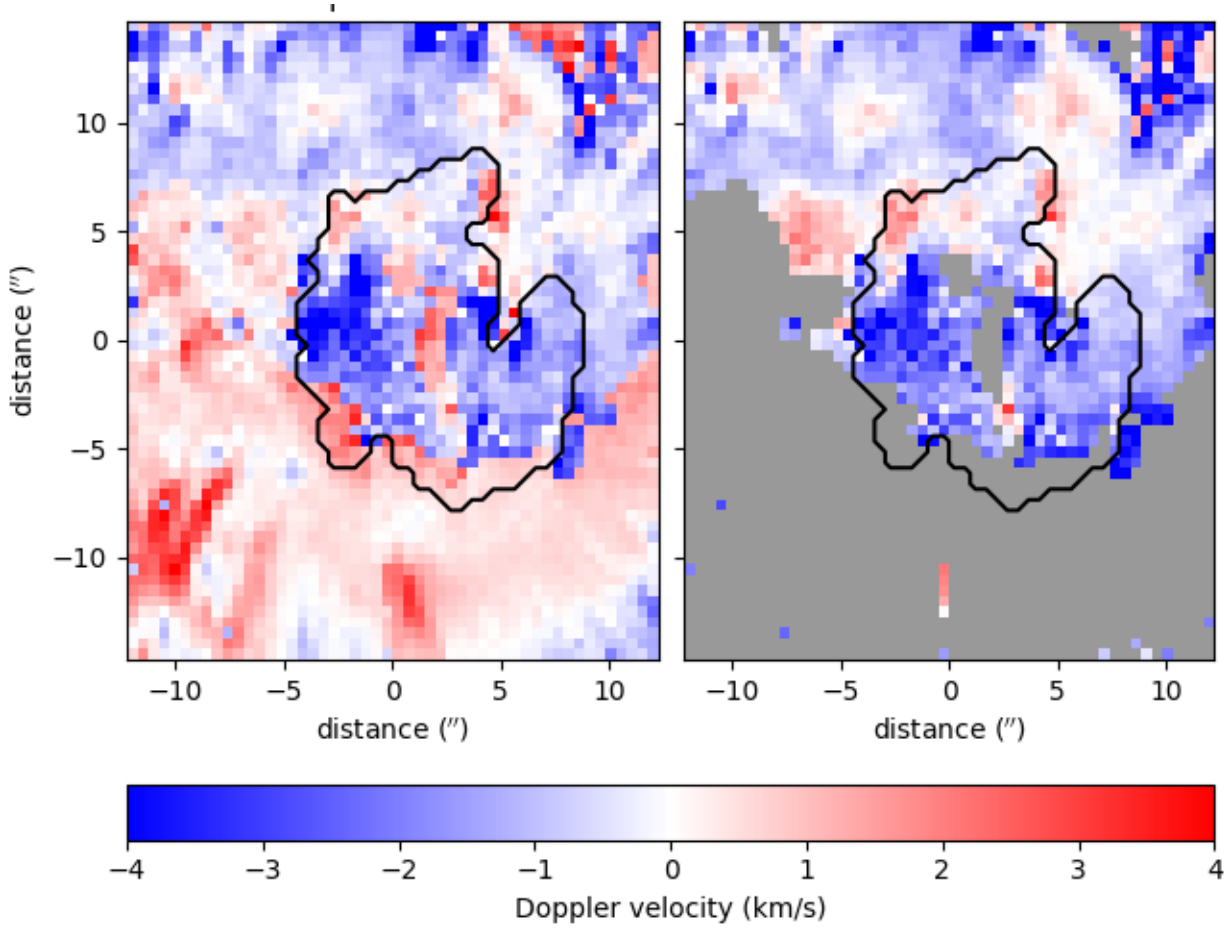
You can calculate and plot Doppler velocities for both the quiescent and active regimes as follows, (with an outline of the sunspot's umbra),

```
vq = results.velocities(model, vtype='quiescent')
va = results.velocities(model, vtype='active')

umbra_mask = np.full_like(backgrounds, True)
umbra_mask[backgrounds < 1100] = False

fig, ax = plt.subplots(1, 2, sharey=True, constrained_layout=True)

settings = {
    'show_colorbar': False, 'vmax': 4, 'offset': (-25, -30),
    'resolution': (0.098 * 5 * u.arcsec, 0.098 * 5 * u.arcsec),
    'umbra_mask': umbra_mask,
```

(continues on next page)

(continued from previous page)

```
}

im = plot_map(vq, ax=ax[0], **settings)
plot_map(va, ax=ax[1], **settings)

ax[0].set_title('quiescent')
ax[1].set_title('active')
ax[1].set_ylabel('')
fig.colorbar(im, ax=ax, location='bottom', label='Doppler velocity (km/s)')

plt.show()
```



Total running time of the script: (0 minutes 8.456 seconds)

Using IBIS8542Model

This is an example showing how to fit an array of spectra using the *mcalf.models.IBIS8542Model* class, and plot and export the results.

Generate sample data

Create spectral grid

First, we shall generate some random sample data to demonstrate the API. The randomly generated spectra are not intended to be representative of observed spectra numerically, they just have a similar shape.

```
import numpy as np
import matplotlib.pyplot as plt

from mcalf.profiles.voigt import voigt, double_voigt
from mcalf.visualisation import plot_spectrum

# Create demo wavelength grid
w = np.linspace(8541, 8543, 20)
```

We shall generate demo spectra in a 2x3 grid. Half of the spectra will have one spectral component (absorption only) and the other half will have two spectral components (mixed absorption and emission components).

Demo spectral components will be modelled as Voigt functions with randomly generated parameters, each within a set range of values.

```
def v(classification, w, *args):
    """Voigt function wrapper."""
    if classification < 1.5:  # absorption only
        return voigt(w, *args[:4], args[-1])
    return double_voigt(w, *args)  # absorption + emission


def s():
    """Generate random spectra for a 2x3 grid."""
    np.random.seed(0)  # same spectra every time
    p = np.random.rand(9, 6)  # random Voigt parameters
    # 0  1  2  3  4  5  6  7  8  # p index
    # a1 b1 s1 g1 a2 b2 s2 g2 d  # Voigt parameter
    # absorption |emission   |background

    p[0] = 100 * p[0] - 1000  # absorption amplitude
    p[4] = 100 * p[4] + 1000  # emission amplitude

    for i in (1, 5):  # abs. and emi. peak positions
        p[i] = 0.05 * p[i] - 0.025 + 8542

    for i in (2, 3, 6, 7):  # Voigt sigmas and gammas
        p[i] = 0.1 * p[i] + 0.1

    p[8] = 300 * p[8] + 2000  # intensity background constant
```

(continues on next page)

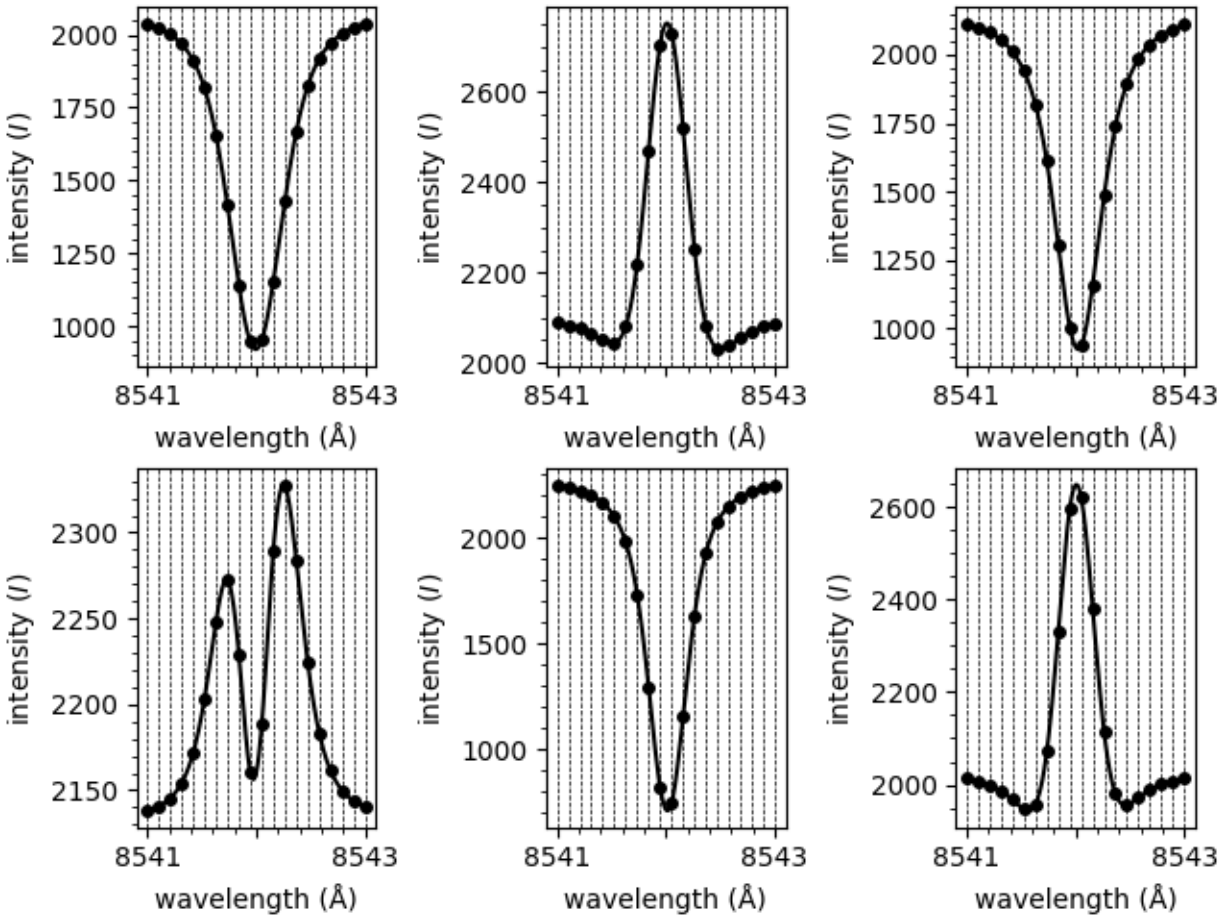
(continued from previous page)

```
    # Define each spectrum's classification
    c = [0, 2, 0, 2, 0, 2]
    # Choose single or double component spectrum
    # based on this inside the function `v()`.

    # Generate the spectra
    specs = [v(c[i], w, *p[:, i]) for i in range(6)]

    # Reshape to 2x3 grid
    return np.asarray(specs).reshape((2, 3, len(w)))


raw_data = s()

print('shape of spectral grid:', raw_data.shape)
```

```
shape of spectral grid: (2, 3, 20)
```

These spectra look as follows,

```
fig, axes = plt.subplots(2, 3, constrained_layout=True)
for ax, spec in zip(axes.flat, raw_data.reshape((6, raw_data.shape[-1]))):

    plot_spectrum(w, spec, normalised=False, ax=ax)

plt.show()
```



Compute the background intensity

MCALF does not model a constant background value, i.e., the fitting process assumes the intensity values far out in the spectrum's wings are zero.

You can however tell *MCALF* what the constant background value is, and it will subtract it from the spectrum before fitting.

This process was not made automatic as we wanted to give the user full control on setting the background value.

For these demo data, we shall simply set the background to the first intensity value of every spectrum. For a real dataset, you may wish to average the background value throughout a range of spectral points, or even do a moving average throughout time. Functions are provided in `mcalf.utils.smooth` to assist with this process.

```
backgrounds = raw_data[:, :, 0]

print('shape of background intensity grid:', backgrounds.shape)
```

```
shape of background intensity grid: (2, 3)
```

Define a demo classifier

In this example we will not demonstrate how to create a neural network classifier. *MCALF* offers a lot of flexibility when it comes to the classifier. This demo classifier outlines the basic API that is required. By default, the model has a `sklearn.neural_network.MLPClassifier` object preloaded for use as the classifier. The methods *`mcalf.models.ModelBase.train()`* and *`mcalf.models.ModelBase.test()`* are provided by *MCALF* to assist with training the neural network. There is also a useful script in the *Getting Started* section under *User Documentation* in the sidebar for semi-automating the process of creating the ground truth dataset.

Please see tutorials for packages such as scikit-learn for more in-depth advice on creating classifiers.

As we only have six spectra with two distinct shapes, we can create a very simple classifier that classifies spectra based on whether their central intensity is greater or smaller than the left most intensity.

```
class DemoClassifier:
    @staticmethod
    def predict(X):
        y = np.zeros(len(X), dtype=int)
        y[X[:, len(X[0]) // 2] > X[:, 0]] = 2
        return y
```

Using `IBIS8542Model` with the generated data

Initialise the model

Everything we have been doing up to this point has been creating the demo data and classifier. Now we can actually create a model and load in the demo data, although the following steps would be identical for a real dataset.

```
from mcalf.models import IBIS8542Model

# Initialise the model with the wavelength grid
model = IBIS8542Model(original_wavelengths=w)

# Load the spectral shape classifier
model.neural_network = DemoClassifier

# Load the array of spectra and background intensities
model.load_array(raw_data, ['row', 'column', 'wavelength'])
model.load_background(backgrounds, ['row', 'column'])
```

Fit the loaded data

We have now fully initialised the model. We can now call methods to fit the model to the loaded spectra. In the following step we fit all the loaded spectra and a 1D list of *`FitResult`* objects is returned.

```
fits = model.fit(row=[0, 1], column=range(3))

print(fits)
```

```
Processing 6 spectra
/home/docs/checkouts/readthedocs.org/user_builds/mcalf/checkouts/v1.0.0/src/mcalf/models/
```

(continues on next page)

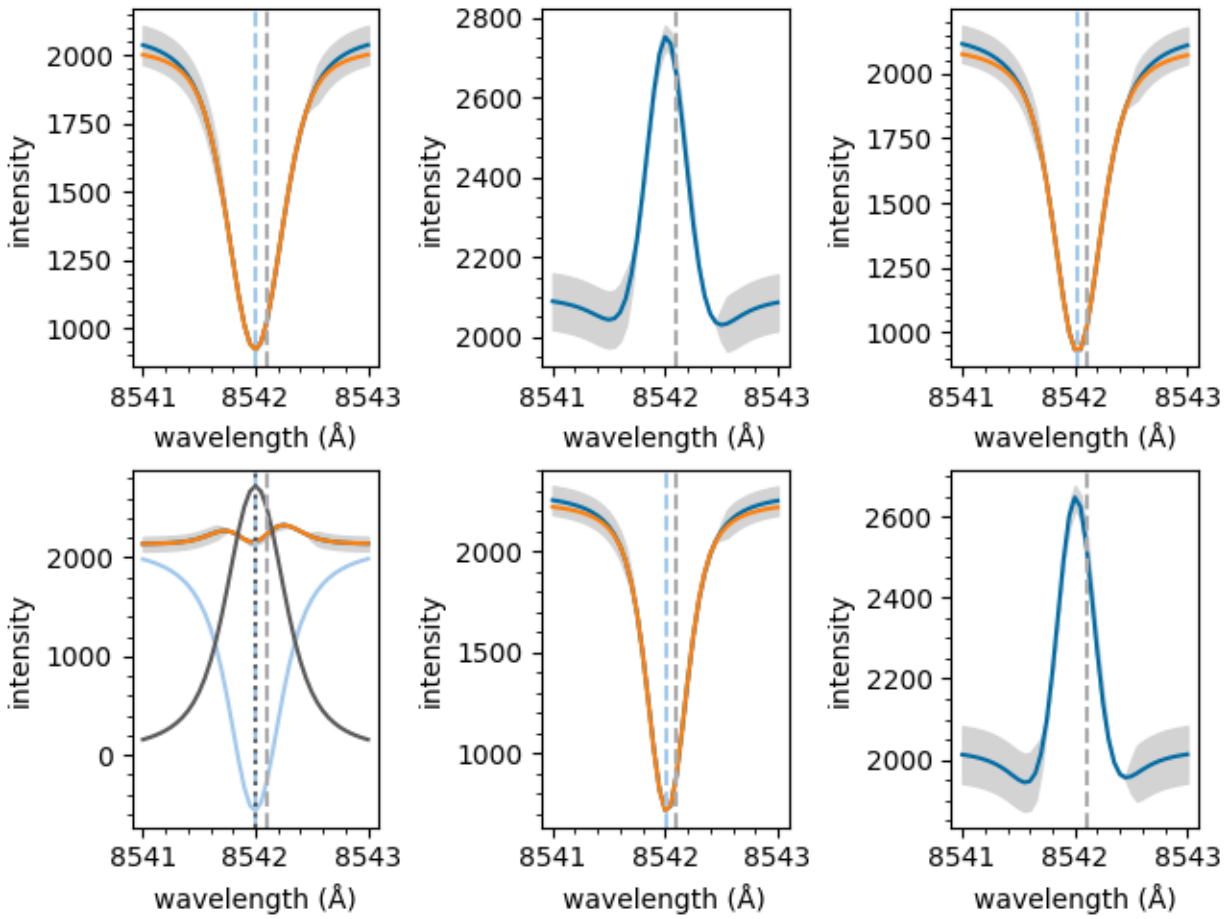
(continued from previous page)

```
↪base.py:1003: UserWarning: RuntimeError(Optimal parameters not found: The maximum␣
↪number of function evaluations is exceeded.) at (0, 0, 1)
  warnings.warn(f"RuntimeError({e}){location_text}")
/home/docs/checkouts/readthedocs.org/user_builds/mcalf/checkouts/v1.0.0/src/mcalf/models/
↪base.py:1003: UserWarning: RuntimeError(Optimal parameters not found: The maximum␣
↪number of function evaluations is exceeded.) at (0, 1, 2)
  warnings.warn(f"RuntimeError({e}){location_text}")
[Successful FitResult with absorption profile of classification 0, Unsuccessful␣
↪FitResult with both profile of classification 2, Successful FitResult with absorption␣
↪profile of classification 0, Successful FitResult with both profile of classification␣
↪2, Successful FitResult with absorption profile of classification 0, Unsuccessful␣
↪FitResult with both profile of classification 2]
```

Plot the fitted parameters

The individual components of each fit can now be plotted separately,

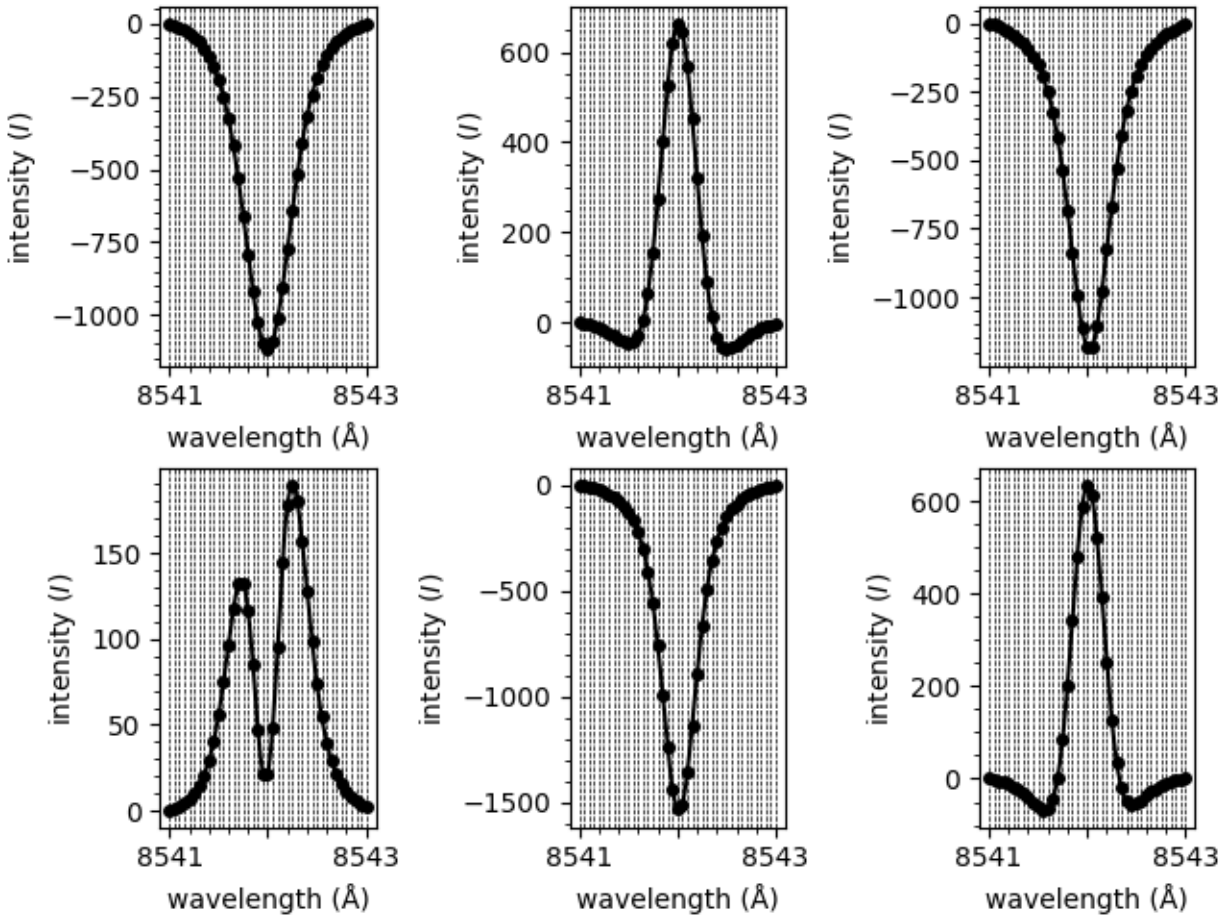
```
fig, axes = plt.subplots(2, 3, constrained_layout=True)
for ax, fit in zip(axes.flat, fits):

    model.plot_separate(fit, show_legend=False, ax=ax)

plt.show()
```



Extract processed spectra

As well as fitting spectra, we can call other methods. In this step we'll extract the array of loaded spectra. However, these spectra have been re-interpolated to a new finer wavelength grid. (This grid can be customised when initialising the model.)

```
spectra = model.get_spectra(row=[0, 1], column=range(3))

print('new shape of spectral grid:', spectra.shape)

spectra_1d = spectra.reshape((6, spectra.shape[-1]))

fig, axes = plt.subplots(2, 3, constrained_layout=True)
for ax, spec in zip(axes.flat, spectra_1d):

    plot_spectrum(model.constant_wavelengths, spec,
                  normalised=False, ax=ax)

plt.show()
```




```
new shape of spectral grid: (1, 2, 3, 41)
```

Classify spectra

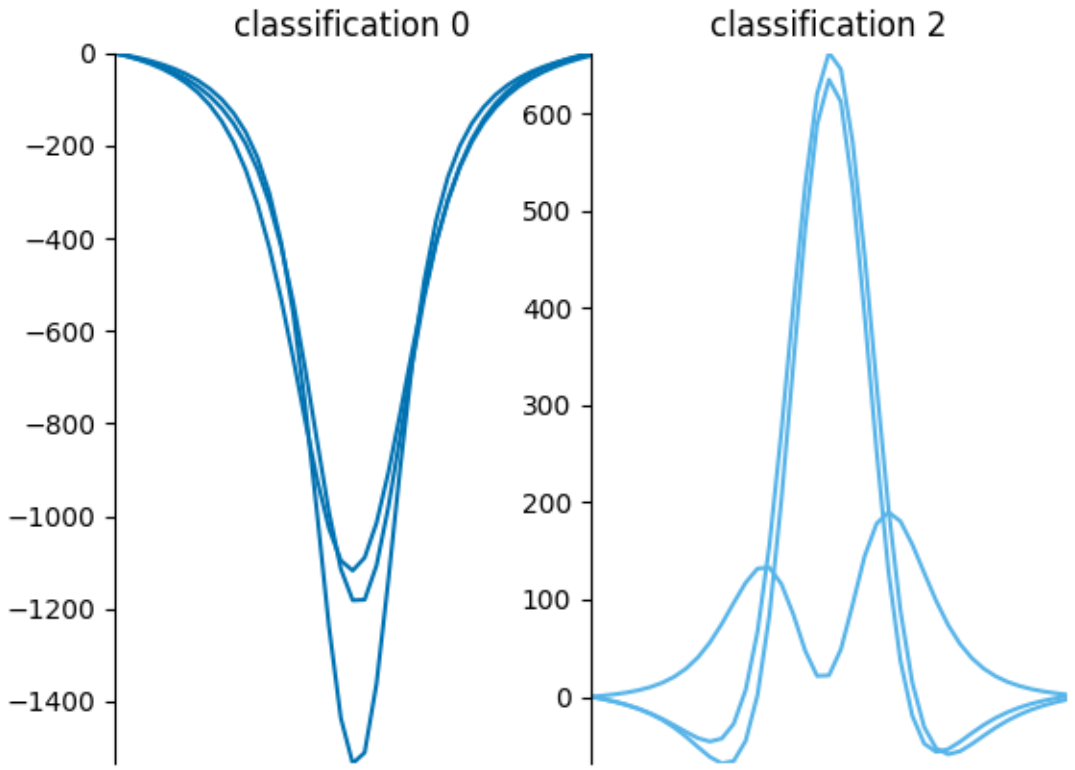
We can also classify the loaded spectra and create plots,

```
classifications = model.classify_spectra(row=[0, 1], column=range(3))

print('classifications are:', classifications)
```

```
classifications are: [[[0 2 0]
  [2 0 2]]]
```

This function plots the spectra grouped by classification,

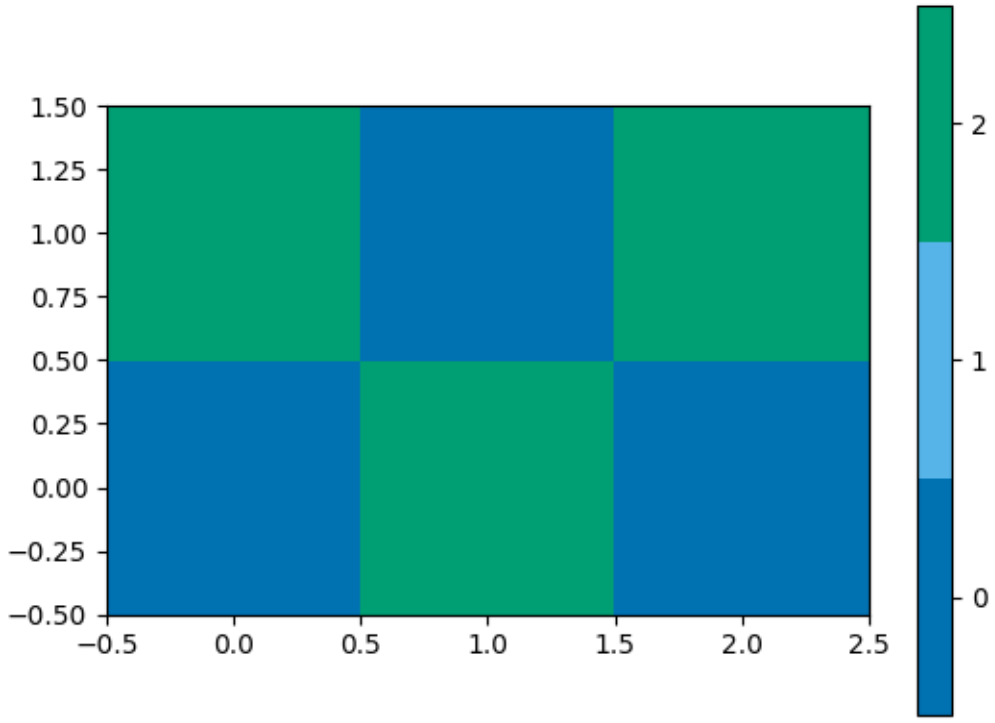
```
from mcalf.visualisation import plot_classifications

plot_classifications(spectra_1d, classifications.flatten())
```




```
GridSpec(1, 2)
```

This function plots a spatial map of the classifications on the 2x3 grid,

```
from mcalf.visualisation import plot_class_map

plot_class_map(classifications)
```




```
<matplotlib.image.AxesImage object at 0x7fe91d8baf40>
```

Merge output data

The *FitResult* objects in the `fits` 1D list can be merged into a grid. Each of these objects can be appended to a single *FitResults* object. This allows for increased data portability,

```
from mcalf.models import FitResults

# Initialise with the grid shape and num. params.
results = FitResults((2, 3), 8)

# Append each fit to the object
for fit in fits:
    results.append(fit)
```

Now that we have appended all 6 fits, we can access the merged data,

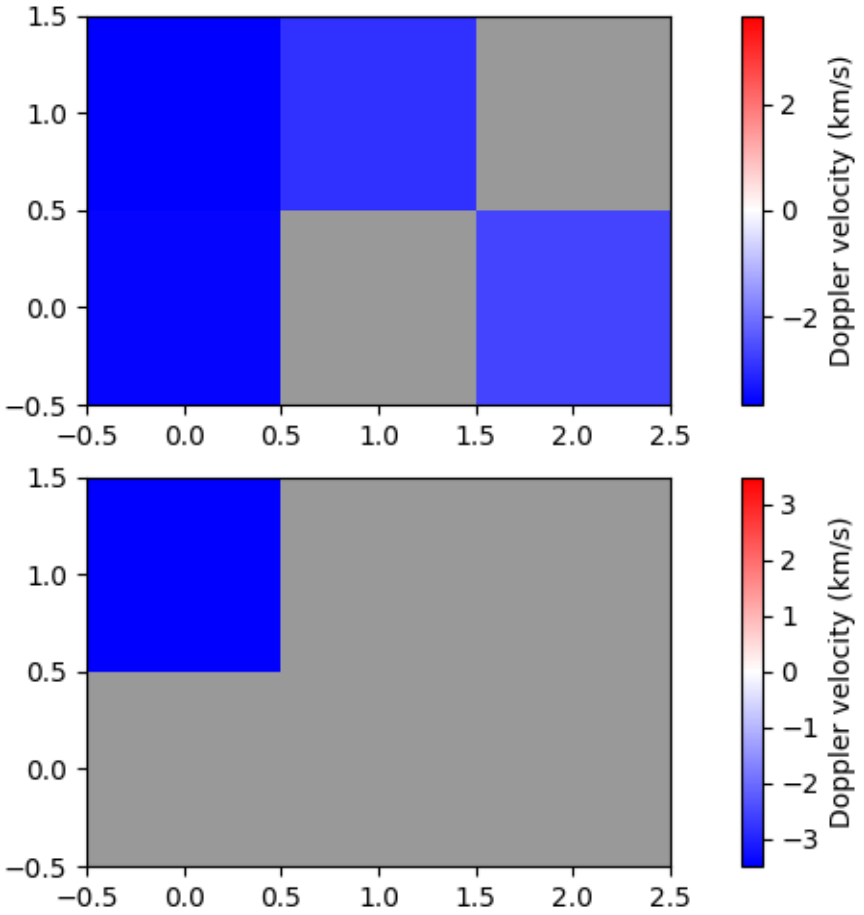
```
print(results.classifications)
print(results.profile)
print(results.success)
print(results.parameters)
```

```
[[0 2 0]
 [2 0 2]]
[['absorption' 'both' 'absorption']
 ['both' 'absorption' 'both']]
[[ True False  True]
 [ True  True False]]
[[[-8.12854071e+02  8.54199697e+03  1.78499712e-01  1.26560705e-01
               nan             nan             nan             nan]
  [            nan             nan             nan             nan
               nan             nan             nan             nan]
  [-8.18417945e+02  8.54202321e+03  1.28952238e-01  1.56493579e-01
               nan             nan             nan             nan]]

 [[-2.21478736e+03  8.54199483e+03  1.06846047e-01  2.21738766e-01
    2.33701044e+03  8.54199968e+03  1.48605642e-01  2.05127799e-01]
  [-8.73401645e+02  8.54201460e+03  1.15790594e-01  1.20586703e-01
               nan             nan             nan             nan]
  [            nan             nan             nan             nan
               nan             nan             nan             nan]]]
```

Calculate velocities

And finally, we can calculate Doppler velocities for both the quiescent (absorption) and active (emission) regimes. (The model needs to be given so the stationary line core wavelength is available.)

```
quiescent = results.velocities(model, vtype='quiescent')
active = results.velocities(model, vtype='active')

from mcalf.visualisation import plot_map

fig, axes = plt.subplots(2, constrained_layout=True)
plot_map(quiescent, ax=axes[0])
plot_map(active, ax=axes[1])
plt.show()
```



Export to FITS file

The *FitResults* object can also be exported to a FITS file,

```
results.save('ibis8542model_demo_fit.fits', model)
```

The file has the following structure,

```
from astropy.io import fits

with fits.open('ibis8542model_demo_fit.fits') as hdul:
    hdul.info()
```

```
Filename: ibis8542model_demo_fit.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      13   (0,)
  1  PARAMETERS    1 ImageHDU        12   (8, 3, 2)   float64
  2  CLASSIFICATIONS    1 ImageHDU        10   (3, 2)   int16
  3  PROFILE       1 ImageHDU        11   (3, 2)   int16
  4  SUCCESS       1 ImageHDU        10   (3, 2)   int16
  5  CHI2          1 ImageHDU        10   (3, 2)   float64
  6  VLOSA         1 ImageHDU        12   (3, 2)   float64
  7  VLOSQ         1 ImageHDU        12   (3, 2)   float64
```

This example outlines the basics of using the *mcalf.models.IBIS8542Model* class. Typically millions of spectra would need to be fitted and processed at the same time. For more details on running big jobs and how spectra can be fitted in parallel, see the *Working with IBIS data* example in the gallery.

Note: Due to limitations with the documentation hosting provider, the examples in the MCALF documentation are computed using a Python based implementation of the Voigt profile instead of the more efficient C version. When this code is run on your own machine it should take much less time than the value reported below.

Total running time of the script: (0 minutes 7.488 seconds)

Visualisation

Below are examples of plots produced using functions within the visualisation module:

Plot a bar chart of classifications

This is an example showing how to produce a bar chart showing the percentage abundance of each classification in a 2D or 3D array of classifications.

First we shall create a random 3D grid of classifications that can be plotted. Usually you would use a method such as *mcalf.models.ModelBase.classify_spectra()* to classify an array of spectra.

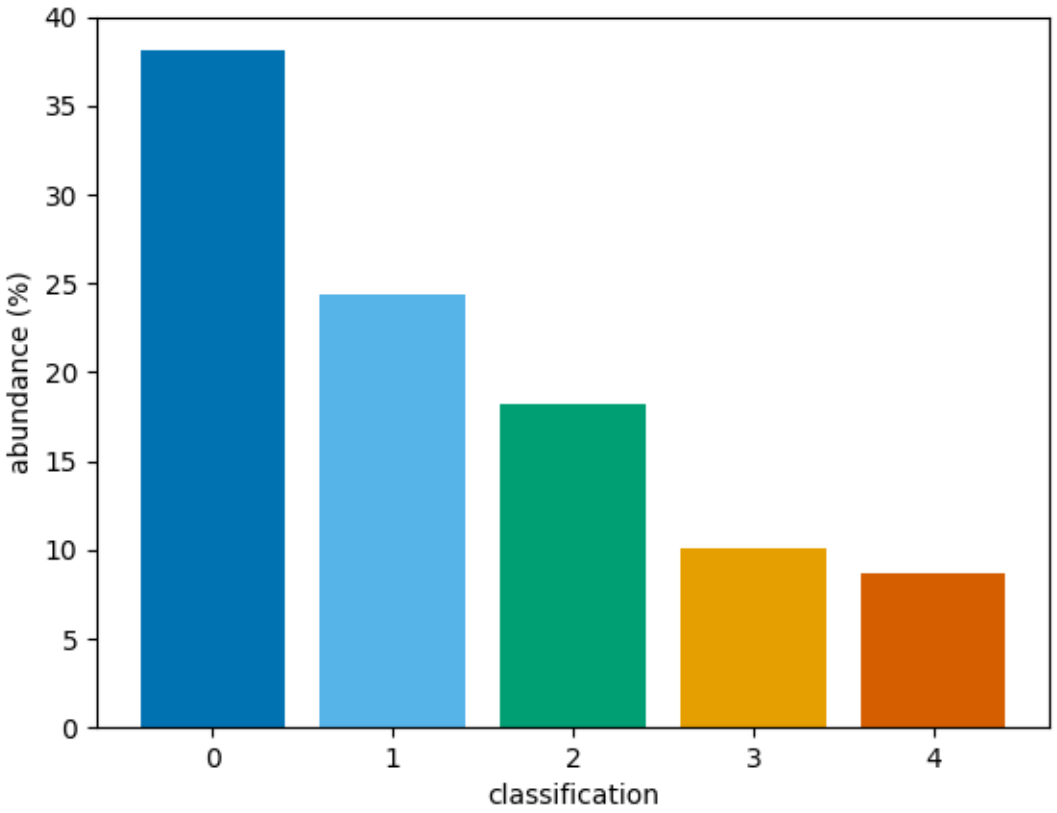
```
from mcalf.tests.helpers import class_map as c

t = 3  # Three images
x = 50  # 50 coordinates along x-axis
y = 20  # 20 coordinates along y-axis
n = 5  # Possible classifications [0, 1, 2, 3, 4]

class_map = c(t, x, y, n)  # 3D array of classifications (t, y, x)
```

Next, we shall import *mcalf.visualisation.bar()*.

```
from mcalf.visualisation import bar
```

We can now simply plot the 3D array. By default, the first dimension of a 3D array will be averaged to produce a time average, selecting the most common classification at each (x, y) coordinate. This means the percentage abundances will correspond to the most common classification at each coordinate.

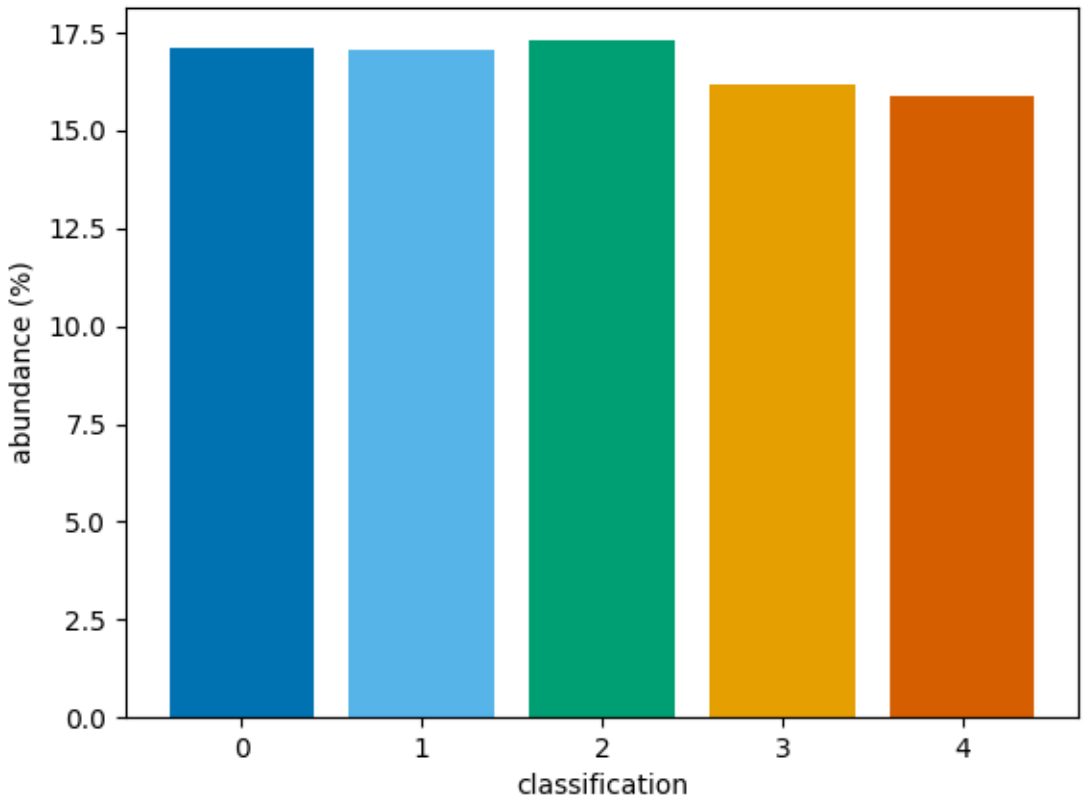
```
bar(class_map)
```




```
<BarContainer object of 5 artists>
```

Instead, the percentage abundances can be determined for the whole 3D array of classifications by setting `reduce=True`. This skips the averaging process.

```
bar(class_map, reduce=False)
```

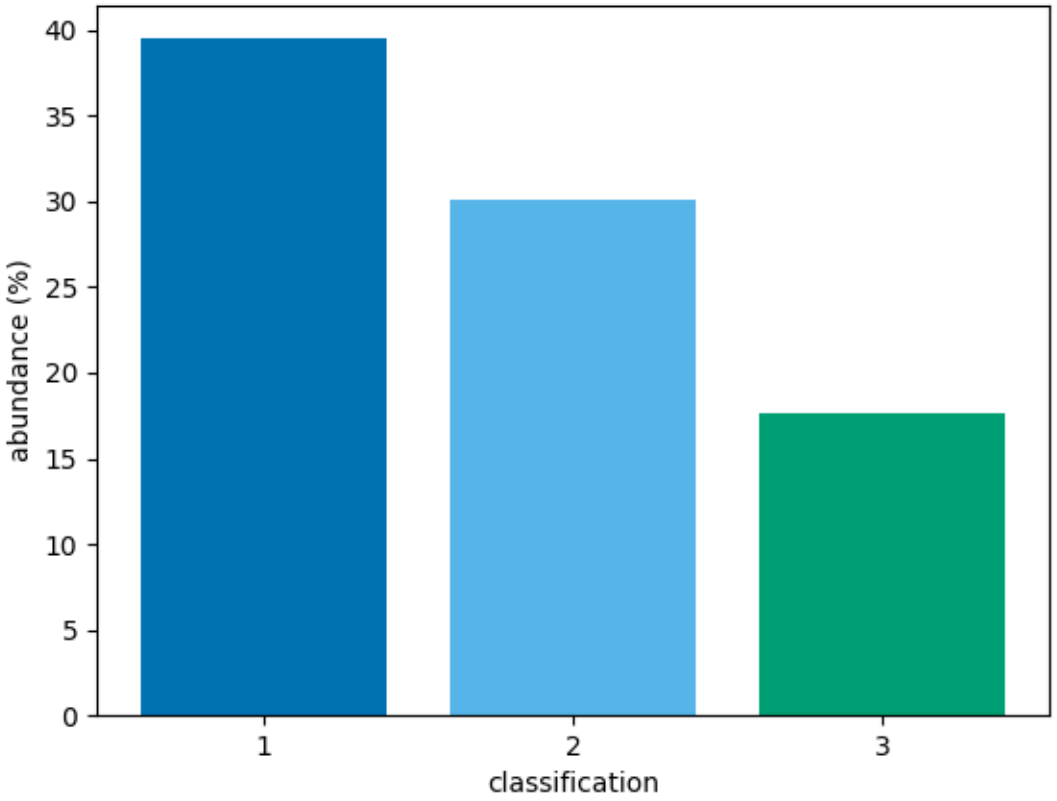



```
<BarContainer object of 5 artists>
```

Alternatively, a 2D array can be passed to the function. If a 2D array is passed, no averaging is needed, and the `reduce` parameter is ignored.

A narrower range of classifications to be plotted can be requested with the `vmin` and `vmax` parameters. To show bars for only classifcations 1, 2, and 3,

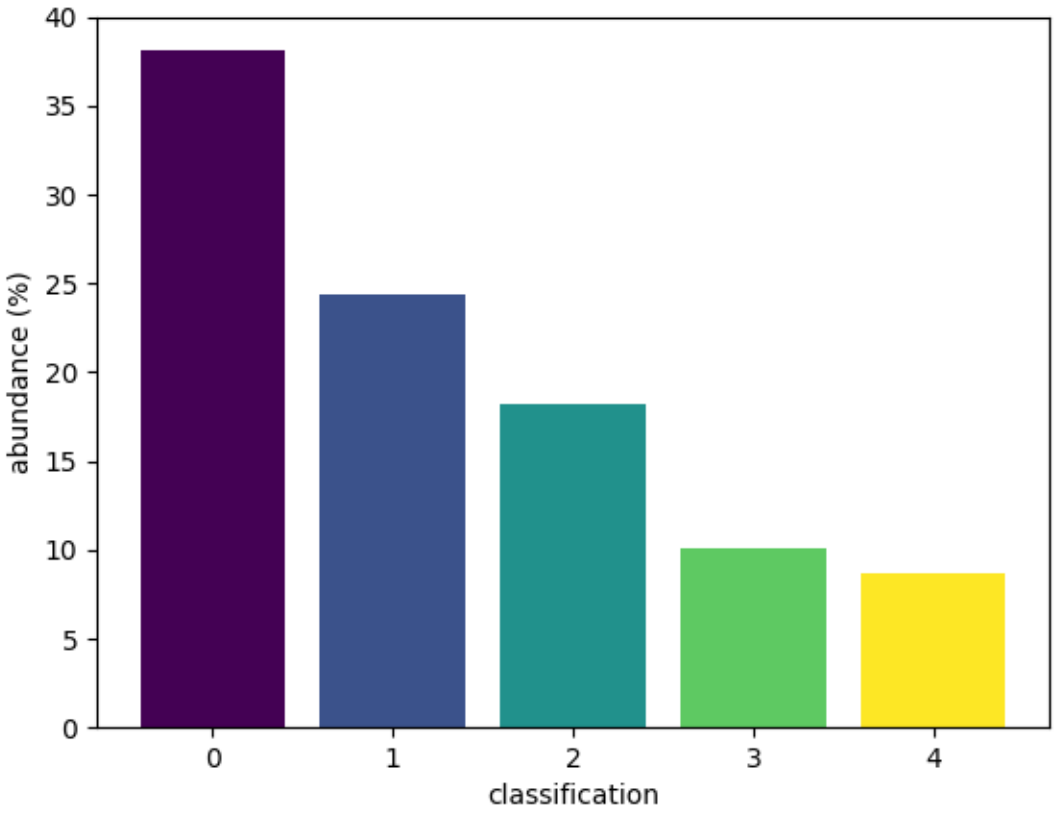
```
bar(class_map, vmin=1, vmax=3)
```




```
<BarContainer object of 3 artists>
```

An alternative set of colours can be requested. Passing a name of a matplotlib colormap to the `style` parameter will produce a corresponding list of colours for each of the bars. For advanced use, explore the `cmap` parameter.

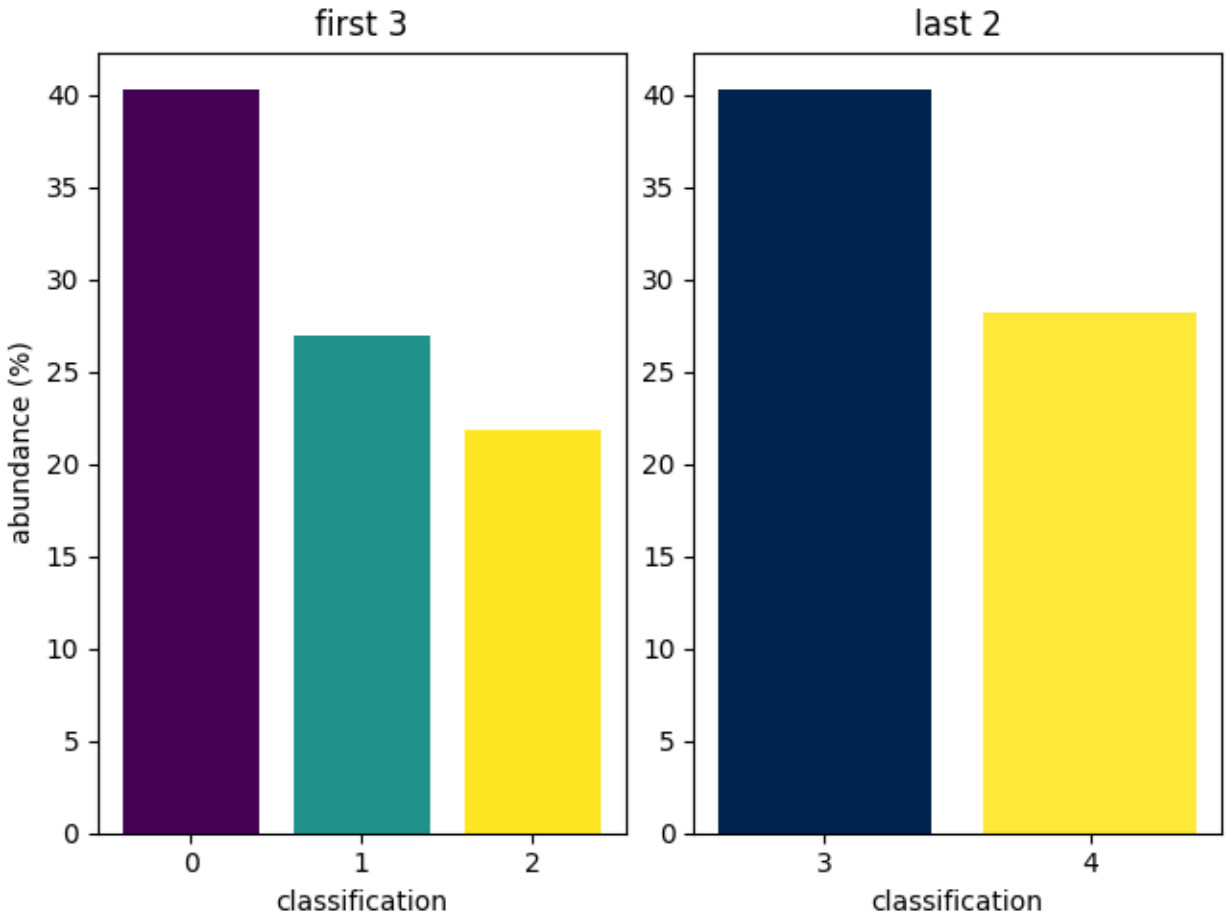
```
bar(class_map, style='viridis')
```




```
<BarContainer object of 5 artists>
```

The bar function integrates well with matplotlib, allowing extensive flexibility.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, constrained_layout=True)

bar(class_map, vmax=2, style='viridis', ax=ax[0])
bar(class_map, vmin=3, style='cividis', ax=ax[1])

ax[0].set_title('first 3')
ax[1].set_title('last 2')

ax[1].set_ylabel('')

plt.show()
```



Note that `vmin` and `vmax` are applied before the `reduce` value is applied. So setting these ranges can change the calculated abundances for other classifications if `class_map` is 3D and `reduce=True`.

The bars do not add up to 100% as a bar for negative, invalid classifications (and therefore classifications out of the `vmin` and `vmax` range) is not shown.

Total running time of the script: (0 minutes 1.079 seconds)

Combine multiple classification plots

This is an example showing how to use multiple classification plotting functions in a single figure.

First we shall create a random 3D grid of classifications that can be plotted. Usually you would use a method such as *mcalf.models.ModelBase.classify_spectra()* to classify an array of spectra.

```
from mcalf.tests.helpers import class_map as c

t = 1  # One image
x = 20  # 20 coordinates along x-axis
y = 20  # 20 coordinates along y-axis
n = 3  # Possible classifications [0, 1, 2]

class_map = c(t, x, y, n)  # 3D array of classifications (t, y, x)
```

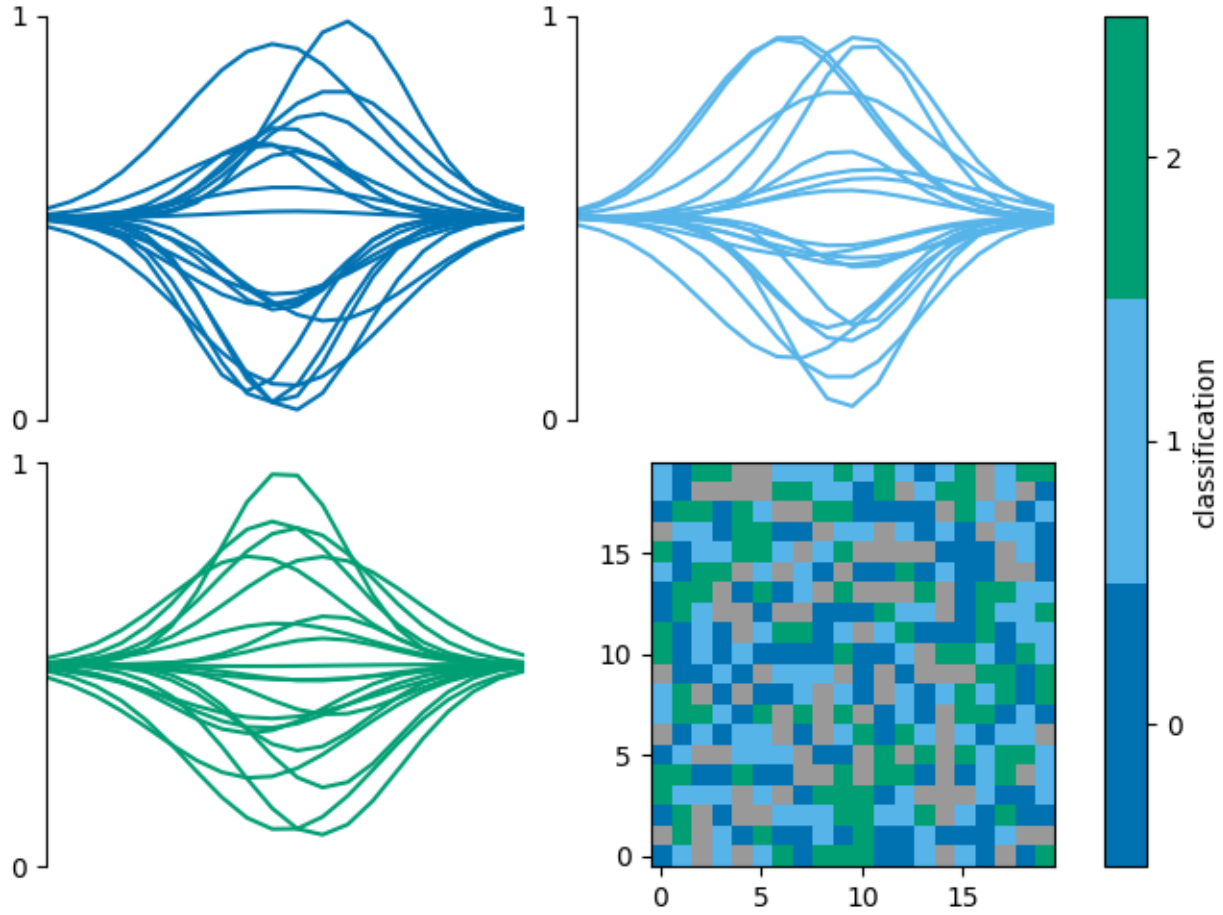
Next we shall create a random array of spectra each labelled with a random classifications. Usually you would provide

your own set of hand labelled spectra taken from spectral imaging observations of the Sun. Or you could provide a set of spectra labelled by the classifier.

```
from mcalf.tests.visualisation.test_classifications import spectra as s

n = 400  # 200 spectra
w = 20  # 20 wavelength points for each spectrum
low, high = 0, 3  # Possible classifications [0, 1, 2]

# 2D array of spectra (n, w), 1D array of labels (n,)
spectra, labels = s(n, w, low, high)
```

If a GridSpec returned by the plot_classification function has free space, a new axes can be added to the returned GridSpec. We can then request plot_class_map to plot onto this new axes. The colorbar axes can be set to `fig.axes` such that the colorbar takes the full height of the figure, as in this case, its colours are the same as the line plots.

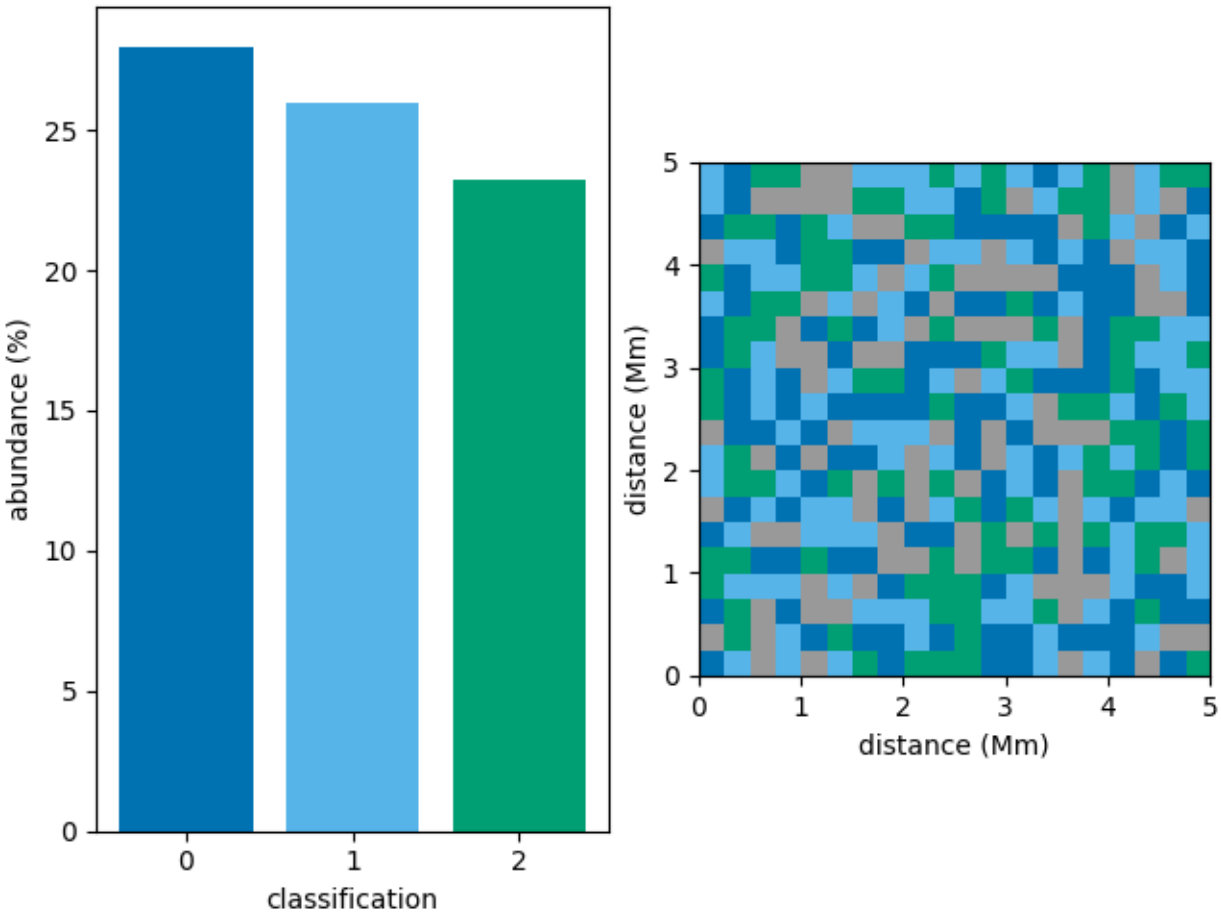
```
import matplotlib.pyplot as plt
from mcalf.visualisation import plot_classifications, plot_class_map

fig = plt.figure(constrained_layout=True)

gs = plot_classifications(spectra, labels, nrows=2, show_labels=False)

ax = fig.add_subplot(gs[-1])
plot_class_map(class_map, ax=ax, colorbar_settings={
    'ax': fig.axes,
    'label': 'classification',
})
```




```
<matplotlib.image.AxesImage object at 0x7fe93794cb20>
```

The function `mcalf.visualisation.init_class_data`() is intended to be an internal function for generating data that is common to multiple plotting functions. However, it may be used externally if necessary.

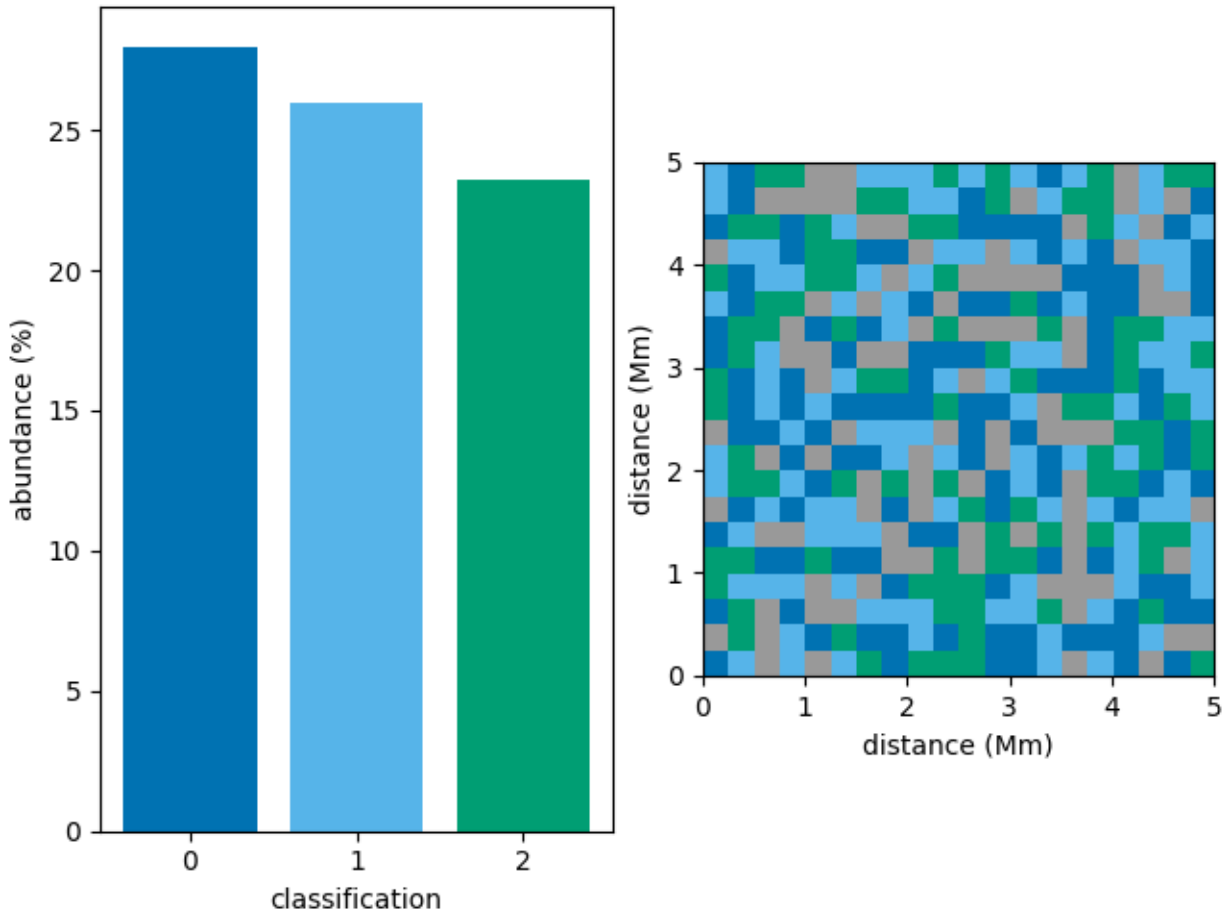
```
from mcalf.visualisation import init_class_data, bar

fig, ax = plt.subplots(1, 2, constrained_layout=True)

data = init_class_data(class_map, resolution=(0.25, 0.25), ax=ax[1])

bar(data=data, ax=ax[0])
plot_class_map(data=data, ax=ax[1], show_colorbar=False)
```




```
<matplotlib.image.AxesImage object at 0x7fe93697fa30>
```

The following example should be equivalent to the example above,

```
fig, ax = plt.subplots(1, 2, constrained_layout=True)

bar(class_map, ax=ax[0])
plot_class_map(class_map, ax=ax[1], show_colorbar=False,
               resolution=(0.25, 0.25))
```




```
<matplotlib.image.AxesImage object at 0x7fe9366d0970>
```

Total running time of the script: (0 minutes 1.099 seconds)

Plot a map of classifications

This is an example showing how to produce a map showing the spatial distribution of spectral classifications in a 2D region of the Sun.

First we shall create a random 3D grid of classifications that can be plotted. Usually you would use a method such as *mcalf.models.ModelBase.classify_spectra()* to classify an array of spectra.

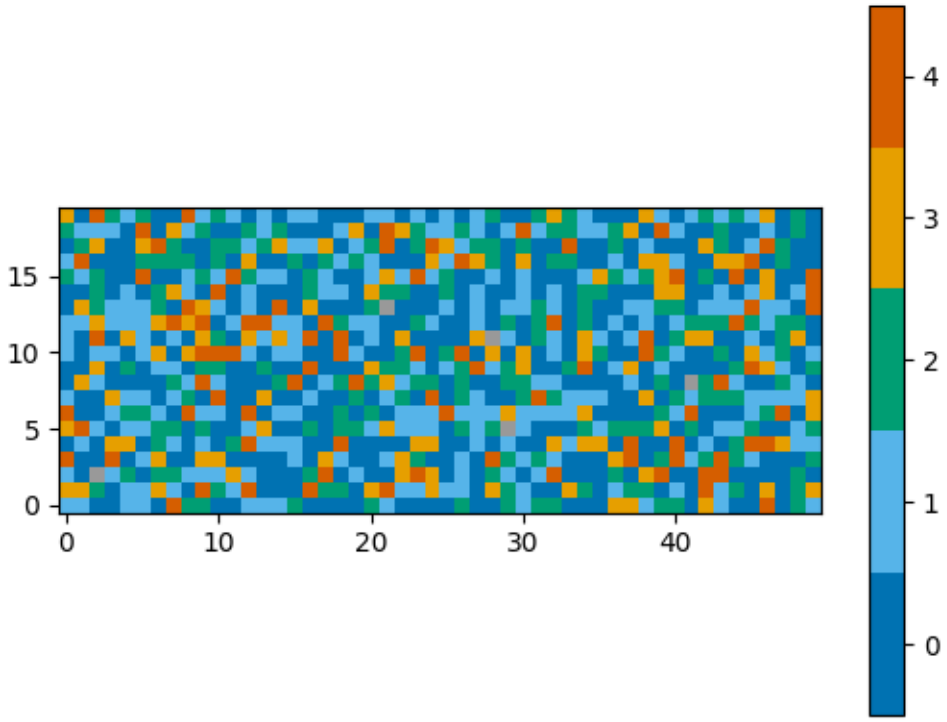
```
from mcalf.tests.helpers import class_map as c

t = 3  # Three images
x = 50  # 50 coordinates along x-axis
y = 20  # 20 coordinates along y-axis
n = 5  # Possible classifications [0, 1, 2, 3, 4]

class_map = c(t, x, y, n)  # 3D array of classifications (t, y, x)
```

Next, we shall import *mcalf.visualisation.plot_class_map()*.

```
from mcalf.visualisation import plot_class_map
```

We can now simply plot the 3D array. By default, the first dimension of a 3D array will be averaged to produce a time average, selecting the most common classification at each (x, y) coordinate.

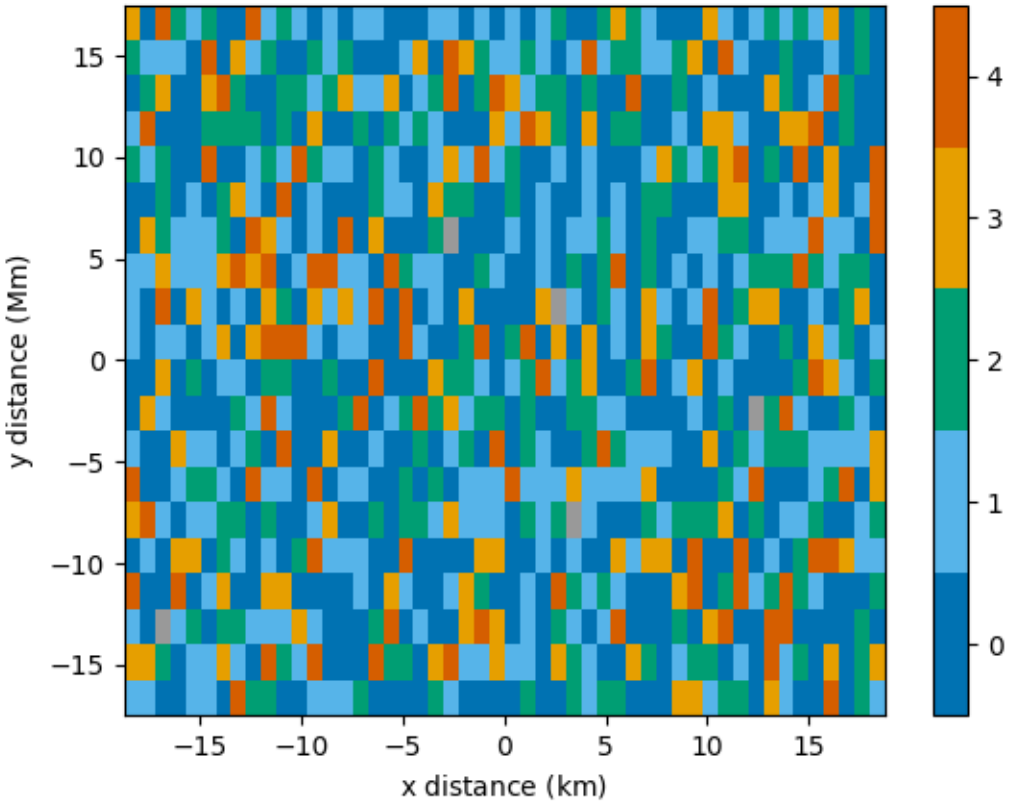
```
plot_class_map(class_map)
```




```
<matplotlib.image.AxesImage object at 0x7fe9355a0a00>
```

A spatial resolution with units can be specified for each axis.

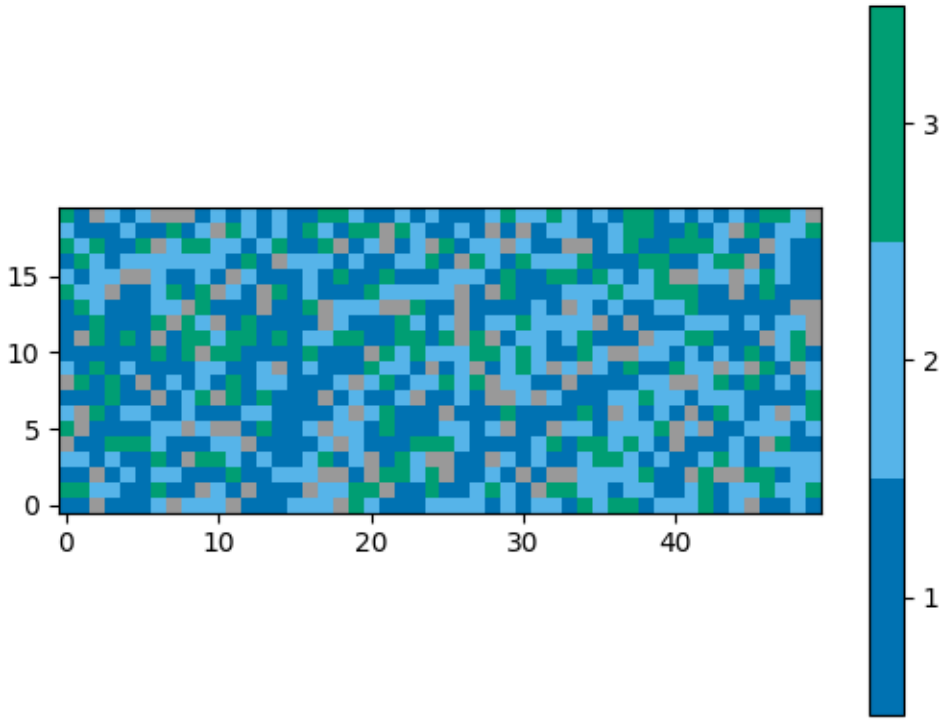
```
import astropy.units as u

plot_class_map(class_map, resolution=(0.75 * u.km, 1.75 * u.Mm),
               offset=(-25, -10),
               dimension=('x distance', 'y distance'))
```




```
<matplotlib.image.AxesImage object at 0x7fe91d7f1fa0>
```

A narrower range of classifications to be plotted can be requested with the `vmin` and `vmax` parameters. Classifications outside of the range will appear as grey, the same as pixels with a negative, unassigned classification.

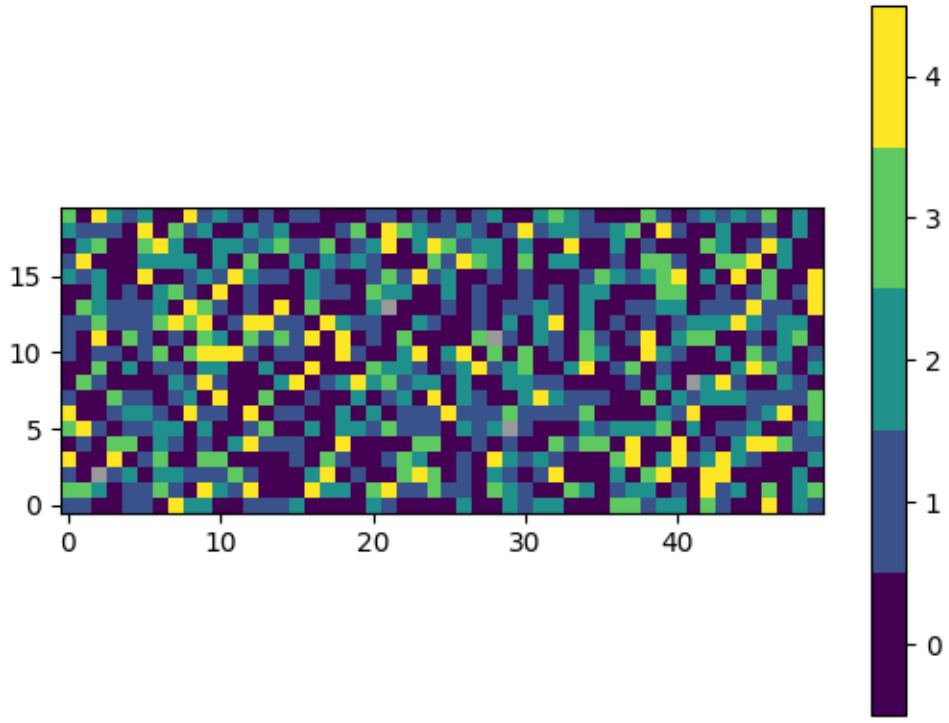
```
plot_class_map(class_map, vmin=1, vmax=3)
```




```
<matplotlib.image.AxesImage object at 0x7fe91db86820>
```

An alternative set of colours can be requested. Passing a name of a matplotlib colormap to the `style` parameter will produce a corresponding list of colours for each of the classifications. For advanced use, explore the `cmap` parameter.

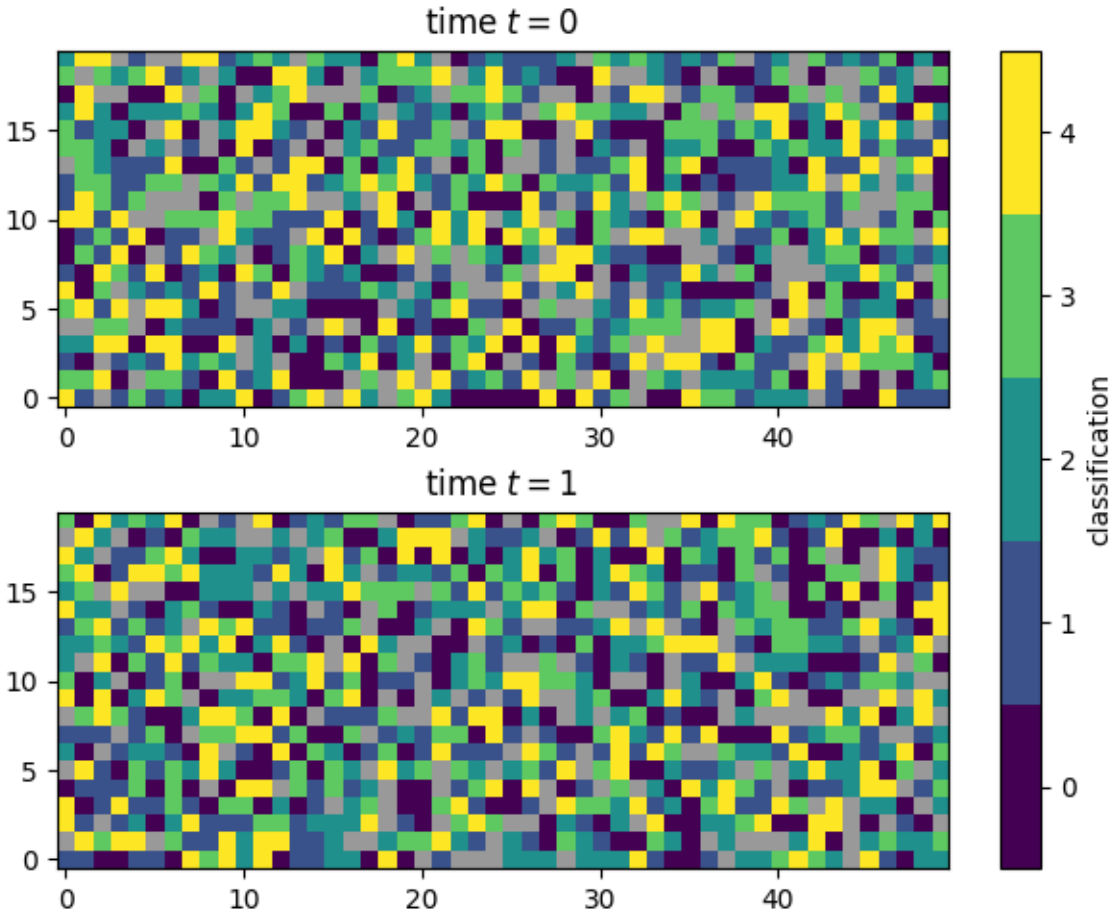
```
plot_class_map(class_map, style='viridis')
```




```
<matplotlib.image.AxesImage object at 0x7fe9356ccdf0>
```

The plot_class_map function integrates well with matplotlib, allowing extensive flexibility. This example also shows how you can plot a 2D `class_map` and skip the averaging.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(2, constrained_layout=True)

plot_class_map(class_map[0], style='viridis', ax=ax[0],
               show_colorbar=False)
plot_class_map(class_map[1], style='viridis', ax=ax[1],
               colorbar_settings={'ax': ax, 'label': 'classification'})

ax[0].set_title('time $t=0$')
ax[1].set_title('time $t=1$')

plt.show()
```



Total running time of the script: (0 minutes 1.223 seconds)

Plot a grid of spectra grouped by classification

This is an example showing how to produce a grid of line plots of an array of spectra labelled with a classification.

First we shall create a random array of spectra each labelled with a random classifications. Usually you would provide your own set of hand labelled spectra taken from spectral imaging observations of the Sun.

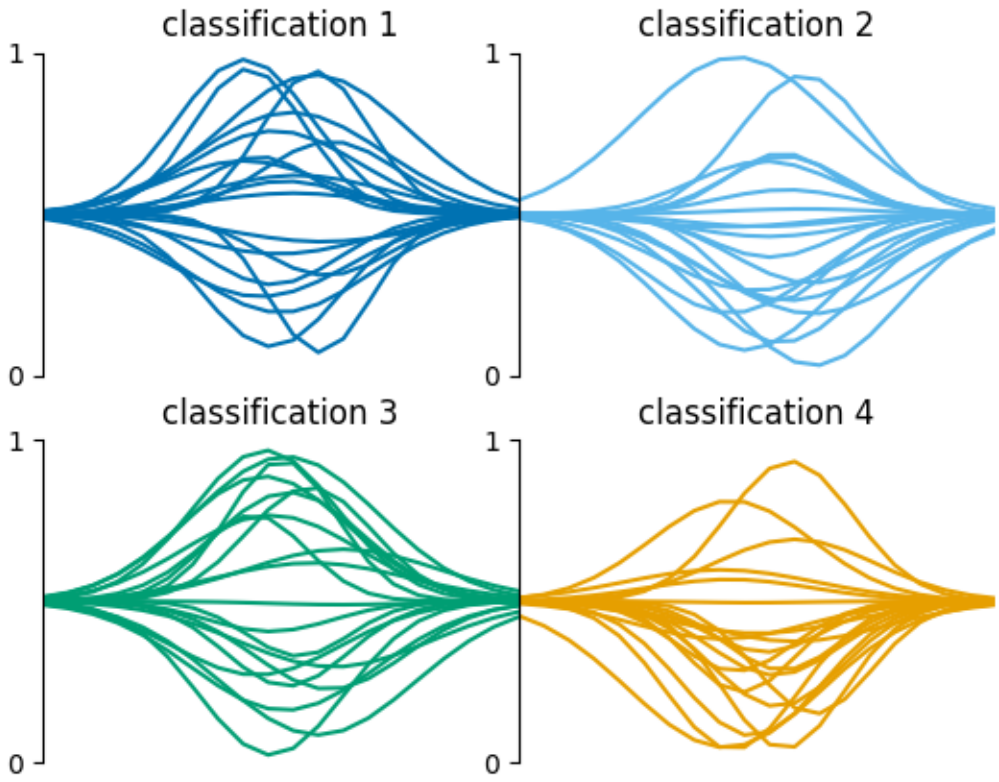
```
from mcalf.tests.visualisation.test_classifications import spectra as s

n = 200  # 200 spectra
w = 20  # 20 wavelength points for each spectrum
low, high = 1, 5  # Possible classifications [1, 2, 3, 4]

# 2D array of spectra (n, w), 1D array of labels (n,)
spectra, labels = s(n, w, low, high)
```

Next, we shall import *mcalf.visualisation.plot_classifications()*.

```
from mcalf.visualisation import plot_classifications
```

We can now plot a simple grid of the spectra grouped by their classification. By default, a maximum of 20 spectra are plotted for each classification. The first 20 spectra are selected.

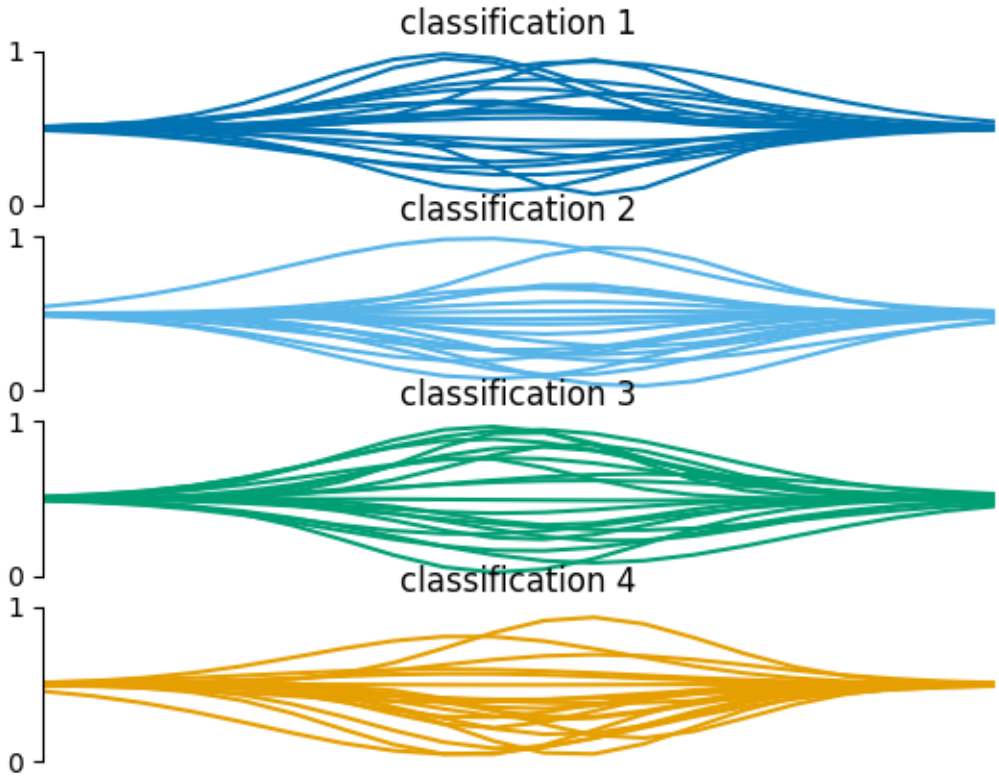
```
plot_classifications(spectra, labels)
```




```
GridSpec(2, 2)
```

A specific number of rows or columns can be requested,

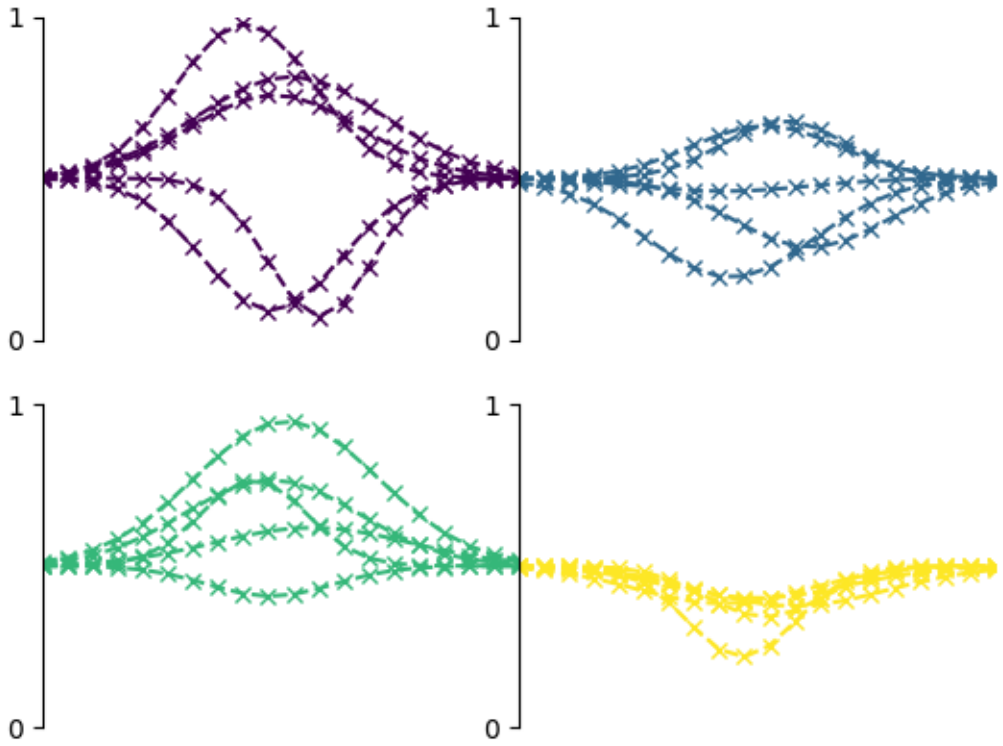
```
plot_classifications(spectra, labels, ncols=1)
```




```
GridSpec(4, 1)
```

The plot settings can be adjusted,

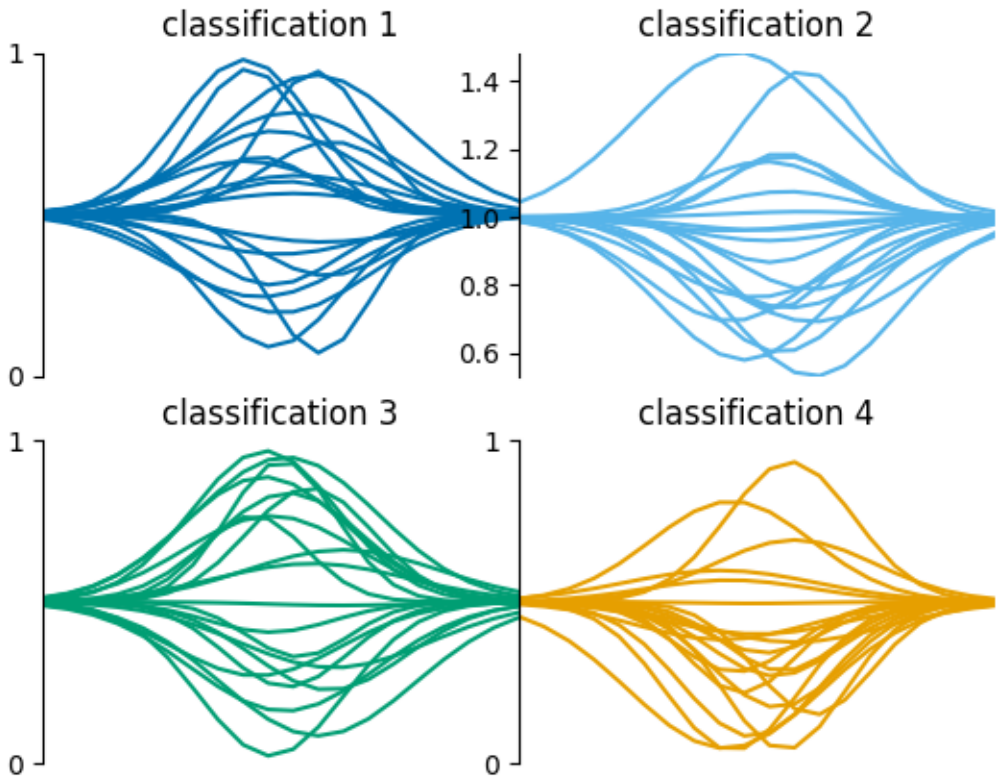
```
plot_classifications(spectra, labels, show_labels=False, nlines=5,
                     style='viridis', plot_settings={'ls': '--', 'marker': 'x'})
```




```
GridSpec(2, 2)
```

By default, the y-axis goes from 0 to 1. This is because labelled training data will typically be rescaled between 0 and 1. However, if a particular classification has spectra that are not within 0 and 1, the y-axis limits are determined by matplotlib.

```
spectra[labels == 2] += 0.5
plot_classifications(spectra, labels)
```




```
GridSpec(2, 2)
```

Total running time of the script: (0 minutes 0.743 seconds)

Plot a fitted spectrum

This is an example showing how to plot the result of fitting a spectrum using the *IBIS8542Model* class.

First we shall create a list of wavelengths, with a variable wavelength spacing. Next, we shall use the Voigt profile to generate spectral intensities at each of the wavelength points. Typically you would provide a spectrum obtained from observations.

The data in this example are produced from randomly selected parameters, so numerical values in this example should be ignored.

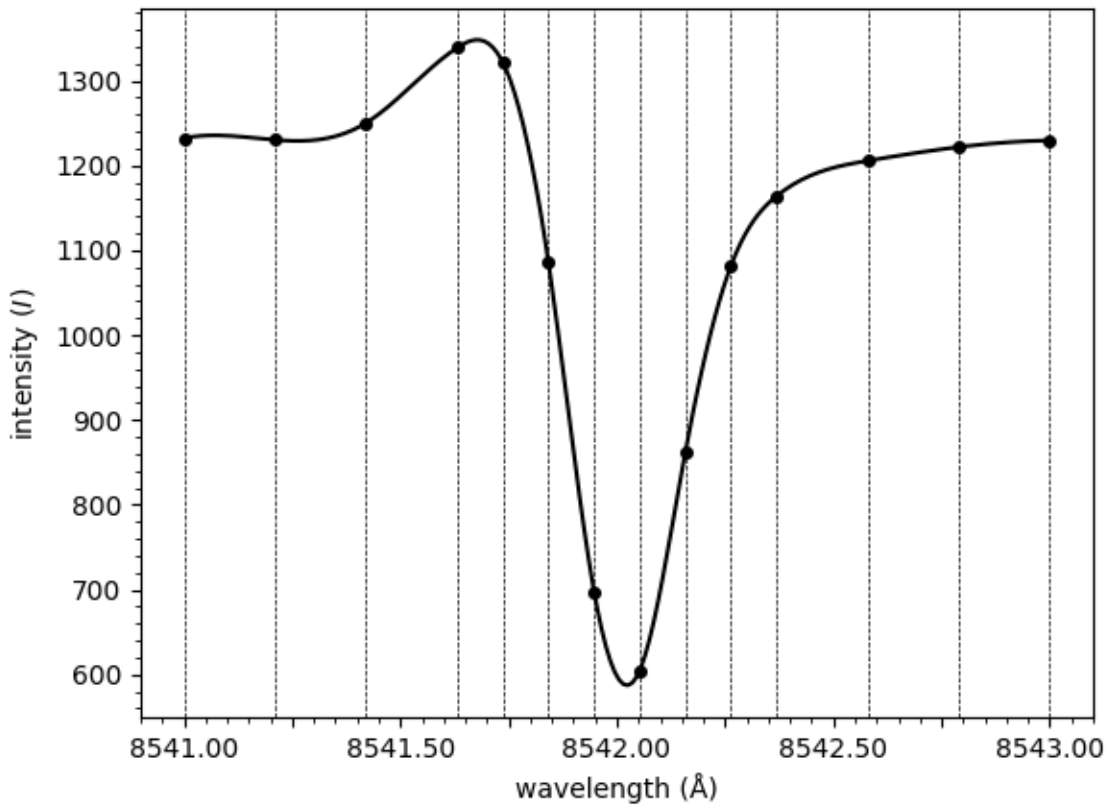
```
import numpy as np
from mcalf.models import IBIS8542Model
from mcalf.profiles.voigt import double_voigt

# Create the wavelength grid and intensity values
wavelengths = np.linspace(8541, 8543, 20)
wavelengths = np.delete(wavelengths, np.s_[1:6:2])
wavelengths = np.delete(wavelengths, np.s_[-6::2])
```

(continues on next page)

(continued from previous page)

```
spectrum = double_voigt(wavelengths, -526, 8542, 0.1, 0.1,
                        300, 8541.9, 0.2, 0.05, 1242)
```

```
from mcalf.visualisation import plot_spectrum

plot_spectrum(wavelengths, spectrum, normalised=False)
```




```
<Axes: xlabel='wavelength (Å)', ylabel='intensity ($I$)'>
```

A basic model is created,

```
model = IBIS8542Model(original_wavelengths=wavelengths)
```

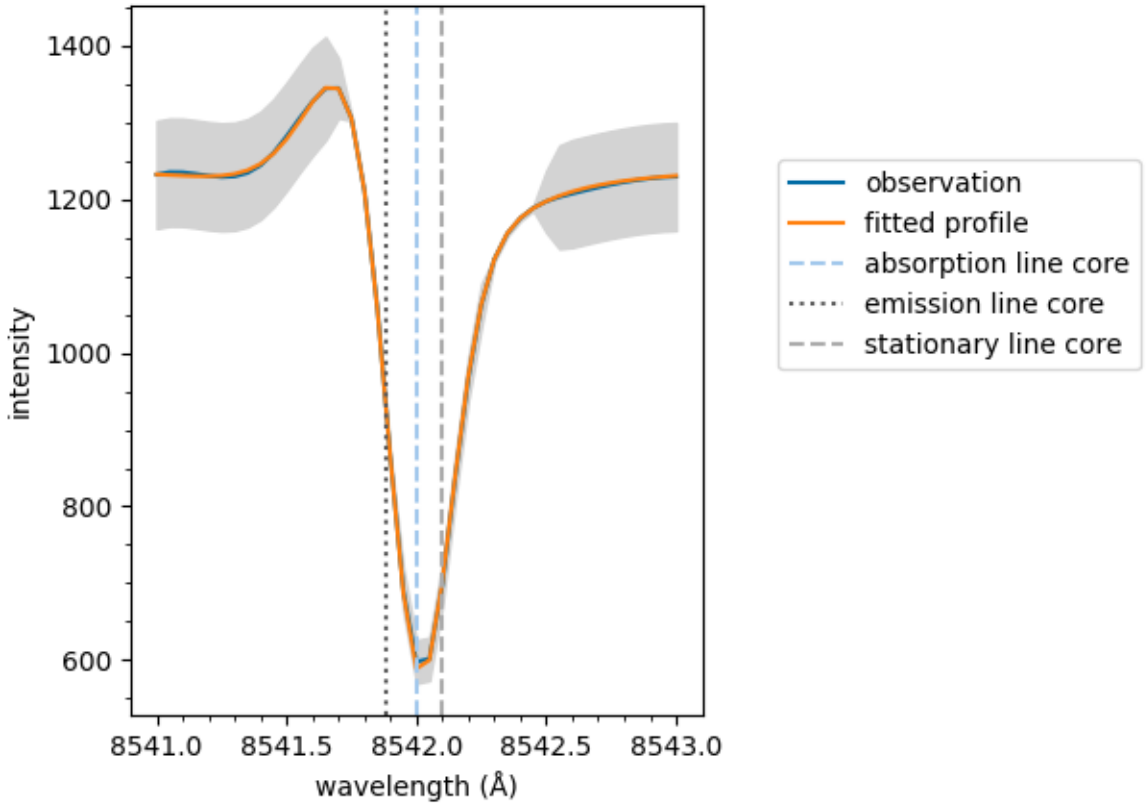
The spectrum is provided to the model's fit method. A classifier has not been loaded so the classification must be provided manually. The fitting algorithm assumes that the intensity at the ends of the spectrum is zero, so in this case we need to provide it with a background value to subtract from the spectrum before fitting.

```
fit = model.fit(spectrum=spectrum, classifications=4, background=1242)

print(fit)
```

```
Successful FitResult with both profile of classification 4
```

The spectrum can now be plotted,

```
model.plot(fit, spectrum=spectrum, background=1242)
```

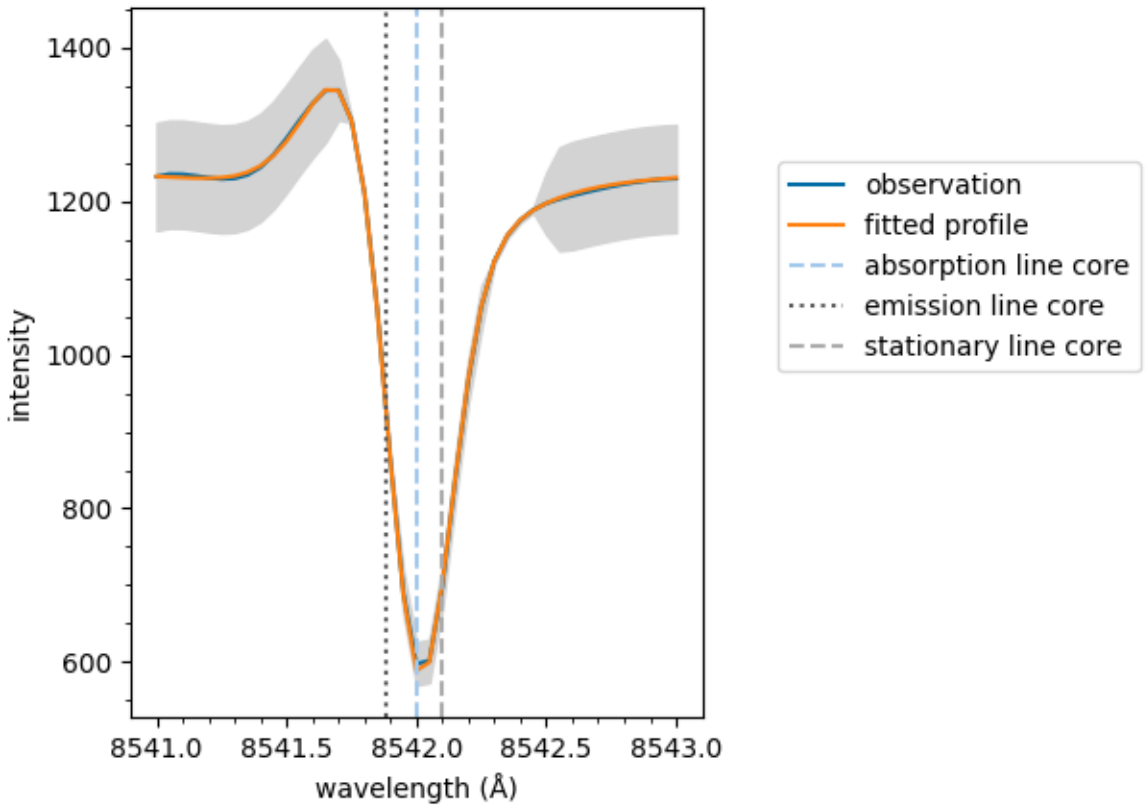



```
<Axes: xlabel='wavelength (Å)', ylabel='intensity'>
```

If an array of spectra and associated background values had been loaded into the model with the *load_array()* and *load_background()* methods respectively, the `spectrum` and `background` parameters would not have to be specified when plotting. This is because the `fit` object would contain indices that the `model` object would use to look up the original loaded values.

Equivalent to above, the plot method can be called on the fit object directly. Remember to specify the `model` which is needed for additional information such as the stationary line core value.

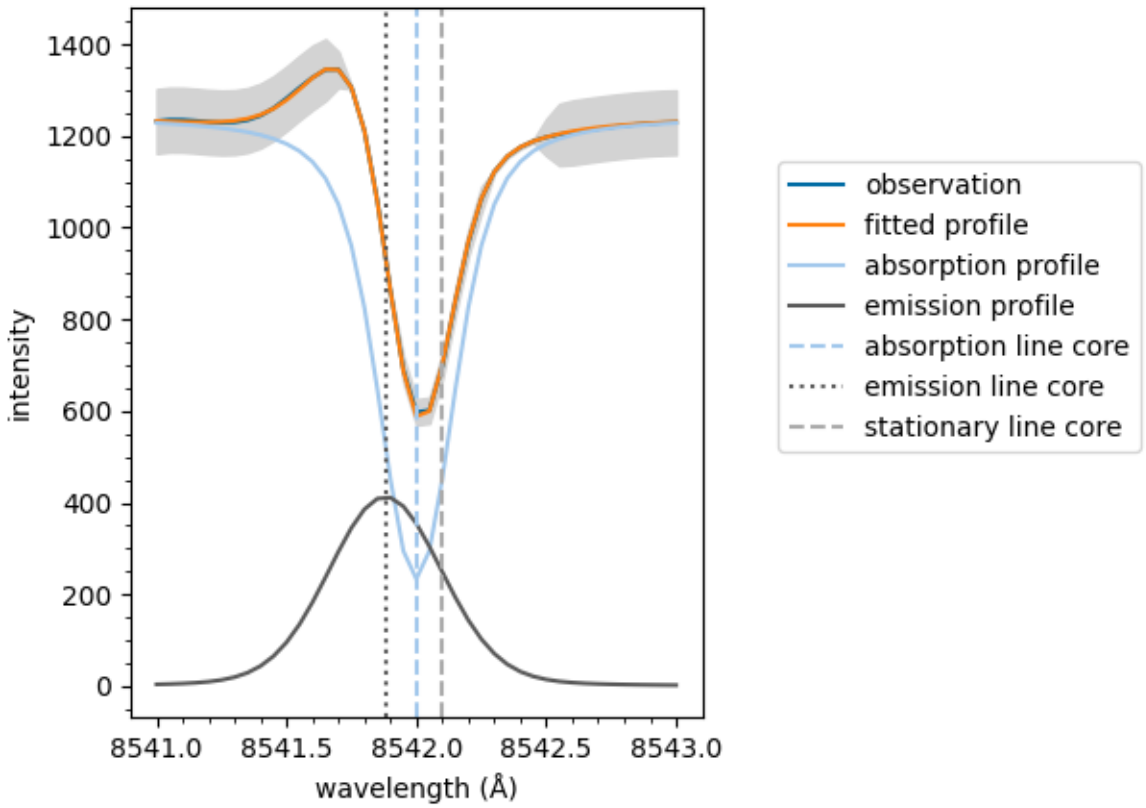
```
fit.plot(model, spectrum=spectrum, background=1242)
```



If the fit has multiple spectral components, such as an active emission profile mixed with a quiescent absorption profile, the follow method can be used to plot the components separatly.

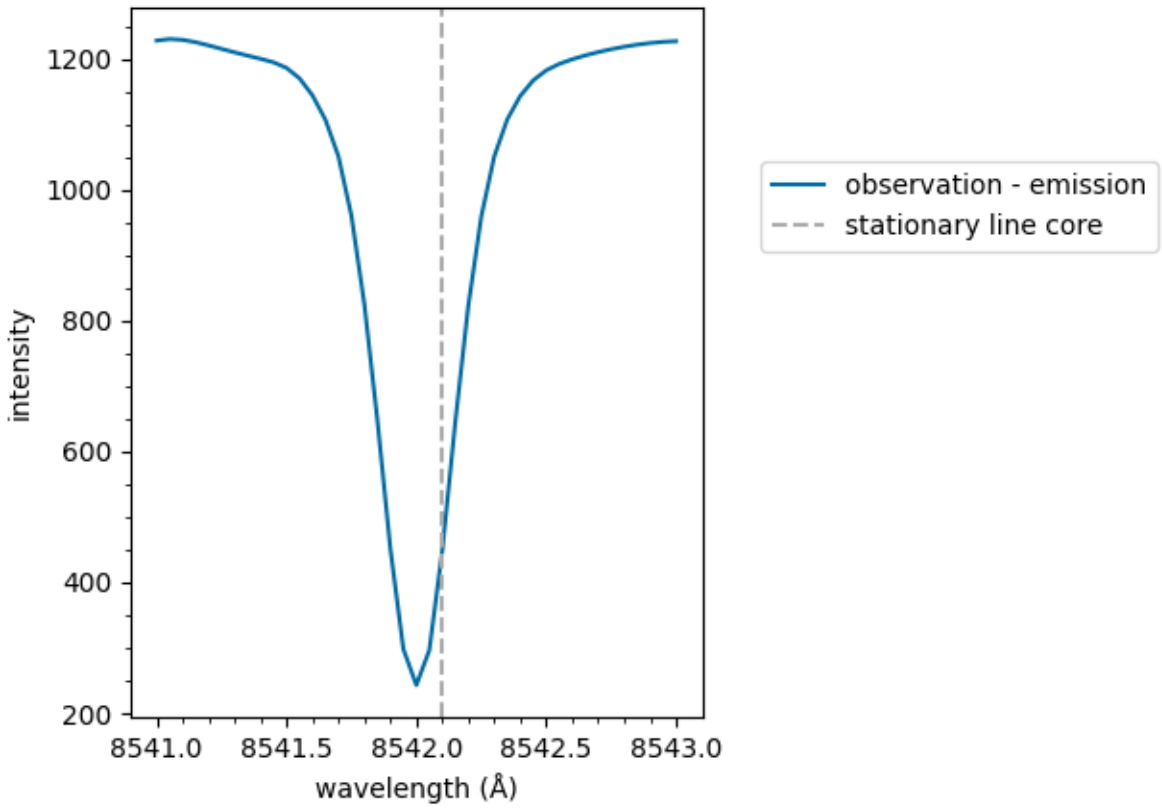
If the fit only has a single component the `plot` method as shown above is used.

```
model.plot_separate(fit, spectrum=spectrum, background=1242)
```



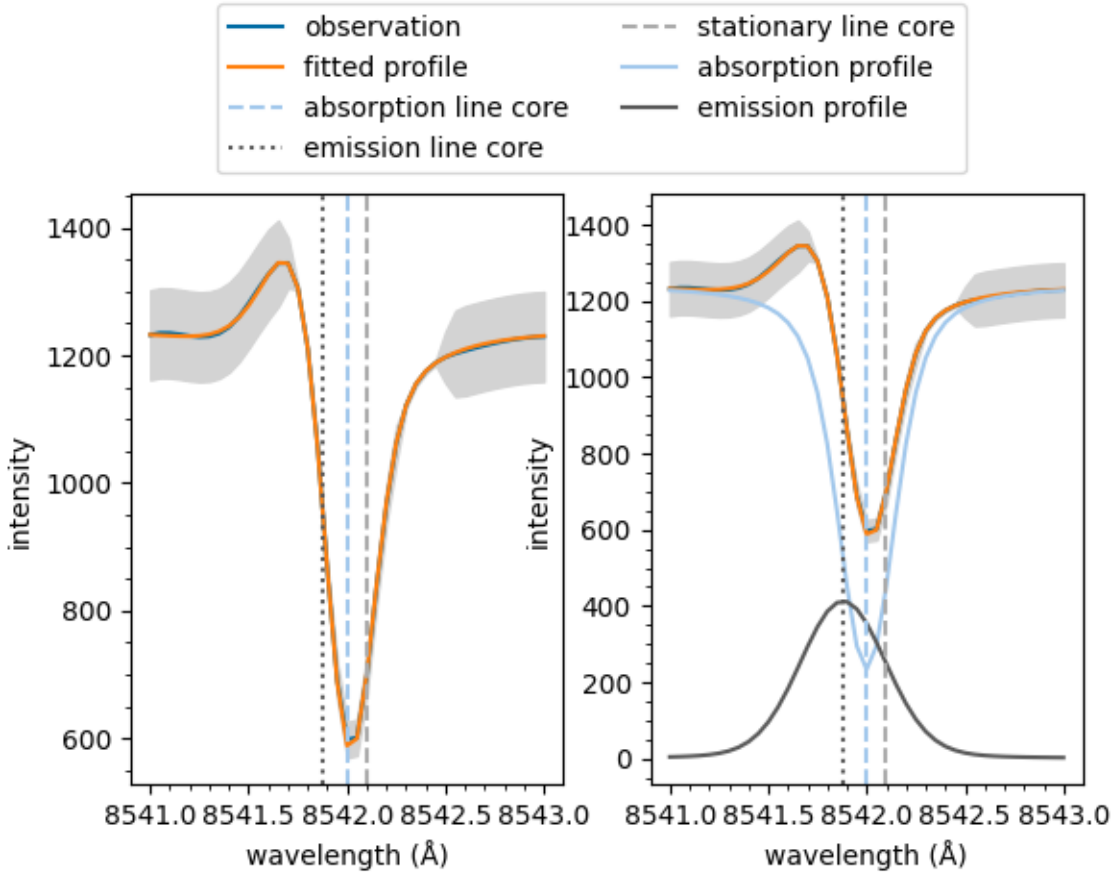
If the fit has an emission component, it is subtracted from the raw spectral data. Otherwise, the default `plot` method is used.

```
model.plot_subtraction(fit, spectrum=spectrum, background=1242)
```



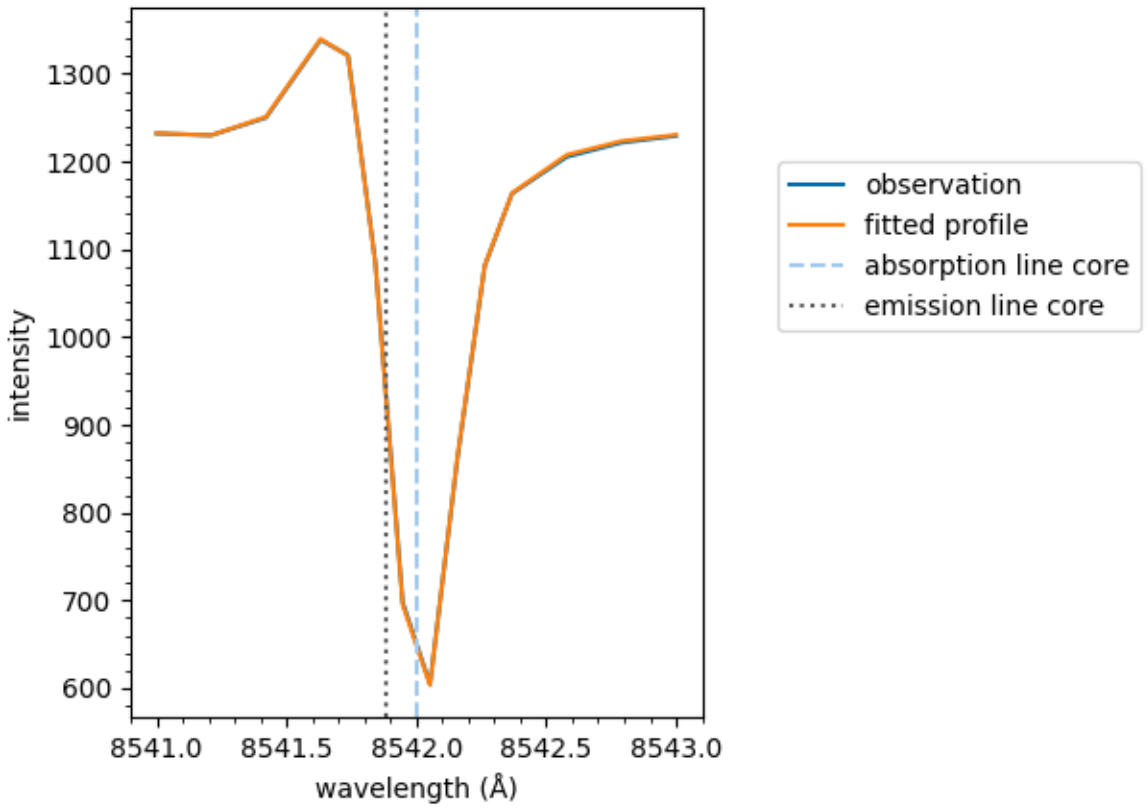
The same line on multiple plots is only labelled the first time it plotted in the figure. This prevents duplicated entries in legends.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2)

model.plot(fit, spectrum=spectrum, background=1242,
           show_legend=False, ax=ax[0])
model.plot_separate(fit, spectrum=spectrum, background=1242,
                    show_legend=False, ax=ax[1])

fig.subplots_adjust(top=0.75)  # Create space above for legend
fig.legend(ncol=2, loc='upper center', bbox_to_anchor=(0.5, 0.97))

plt.show()
```



The underlying *mcalf.visualisation.plot_ibis8542()* function can be used directly. However, it is recommended to plot using the method detailed above as it will do additional processing to the wavelengths and spectrum and also pass additional parameters, such as sigma, to this fitting function.

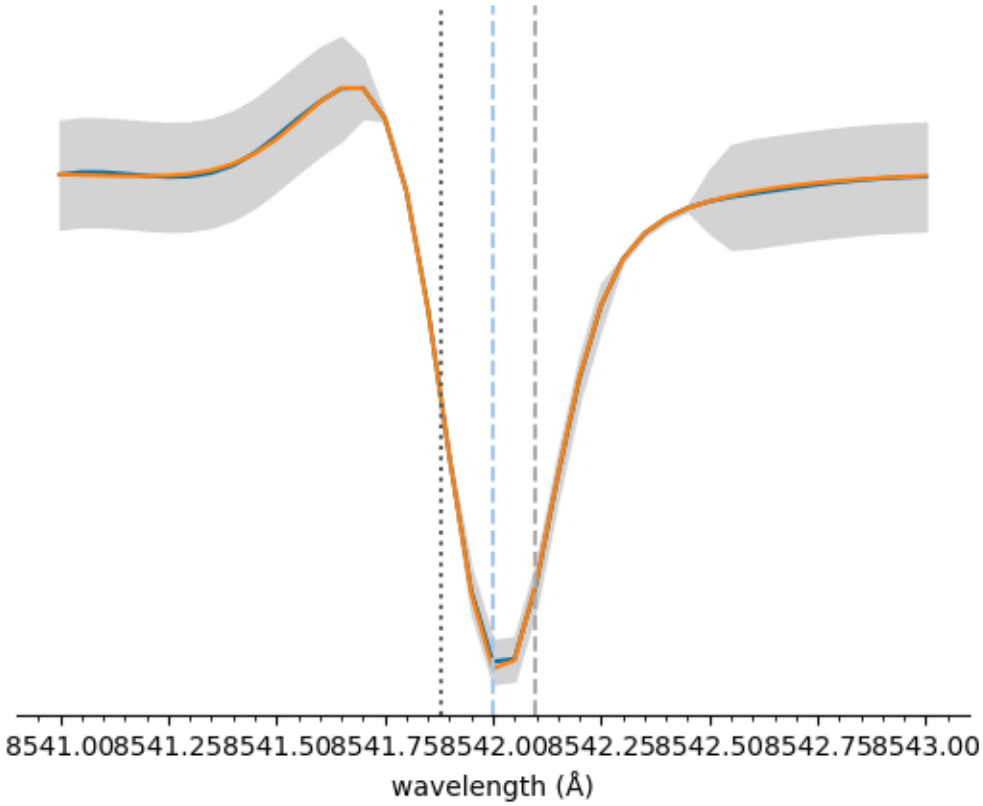
```
from mcalf.visualisation import plot_ibis8542

plot_ibis8542(wavelengths, spectrum, fit.parameters, 1242)
```




```
<Axes: xlabel='wavelength (Å)', ylabel='intensity'>
```

The y-axis and legend can be easily hidden,

```
model.plot(fit, spectrum=spectrum, background=1242,
           show_intensity=False, show_legend=False)
```




```
<Axes: xlabel='wavelength (Å)'>
```

Total running time of the script: (0 minutes 2.082 seconds)

Plot a map of velocities

This is an example showing how to produce a map showing the spatial distribution of velocities in a 2D region of the Sun.

First we shall create a random 2D grid of velocities that can be plotted. Usually you would use a method such as *mcalf.models.FitResults.velocities()* to extract an array of velocities from fitted spectra.

```
import numpy as np
np.random.seed(0)

x = 50  # 50 coordinates along x-axis
y = 40  # 40 coordinates along y-axis
low, high = -10, 10  # range of velocities (km/s)


def a(x, y, low, high):
    arr = np.random.normal(0, (high - low) / 2 * 0.3, (y, x))
    arr[arr < low] = low
```

(continues on next page)

(continued from previous page)

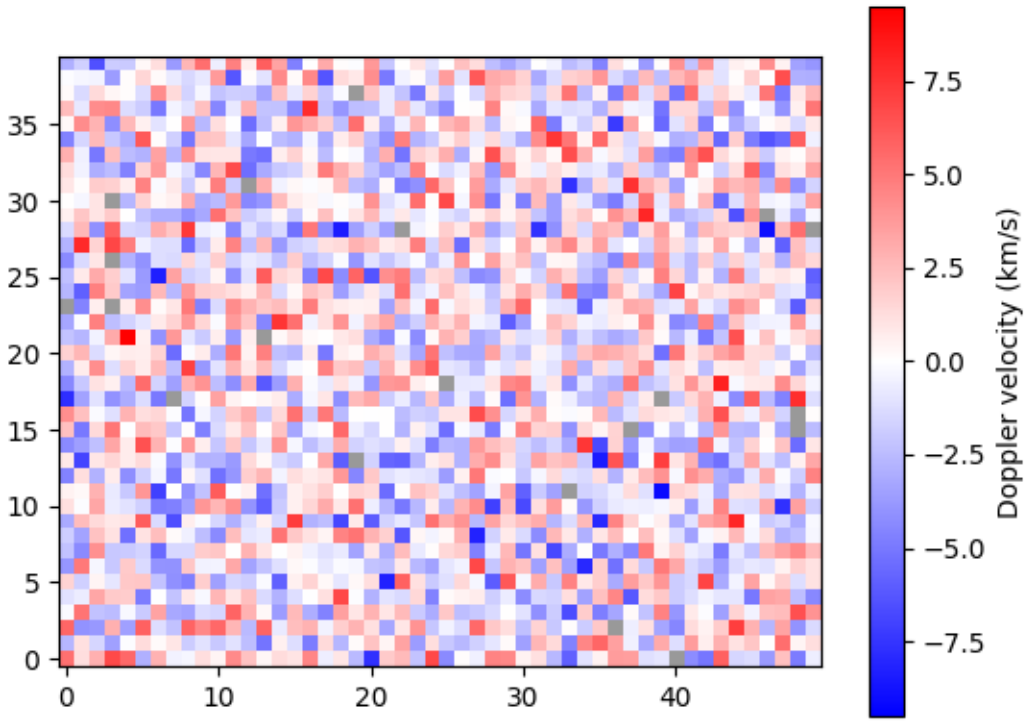
```
    arr[arr > high] = high
    i = np.random.randint(0, arr.size, arr.size // 100)
    arr[np.unravel_index(i, arr.shape)] = np.nan
    return arr


arr = a(x, y, low, high)  # 2D array of velocities (y, x)
```

Next, we shall import *mcalf.visualisation.plot_map()*.

```
from mcalf.visualisation import plot_map
```

We can now simply plot the 2D array.

```
plot_map(arr)
```

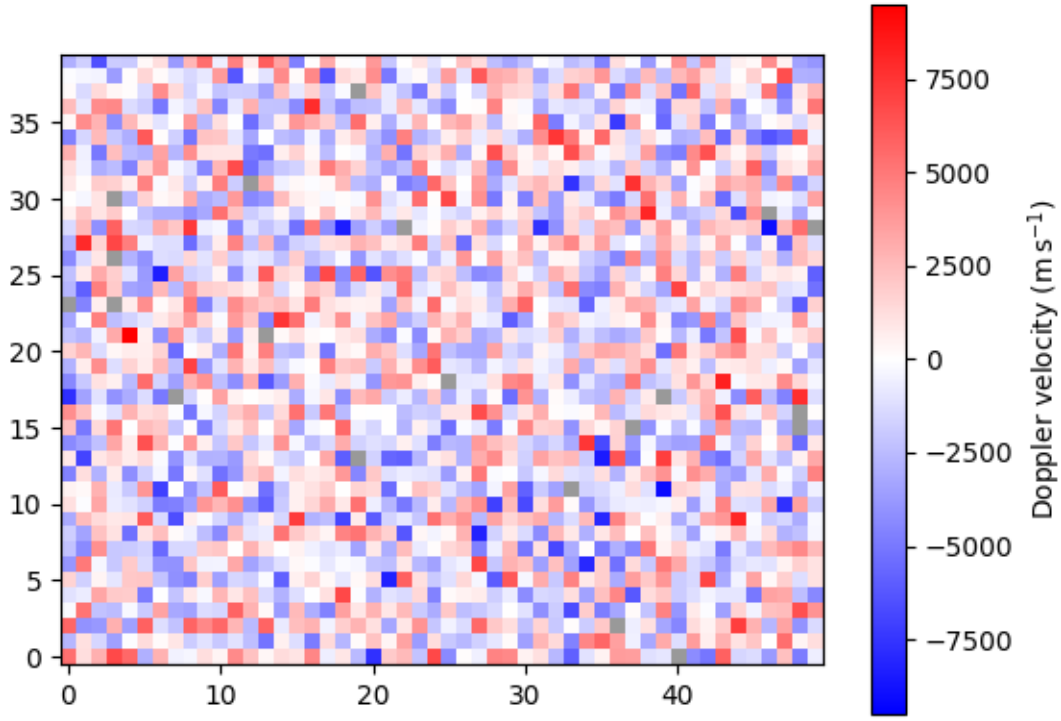



```
<matplotlib.image.AxesImage object at 0x7fe91d957c10>
```

Notice that pixels with missing data (NaN) are shown in grey.

By default, the velocity data are assumed to have units km/s. If your data are not in km/s, you must either 1) rescale the array such that it is in km/s, 2) attach an astropy unit to the array to override the default, or 3) pass an astropy unit to the `unit` parameter to override the default. For example, we can change from km/s to m/s,

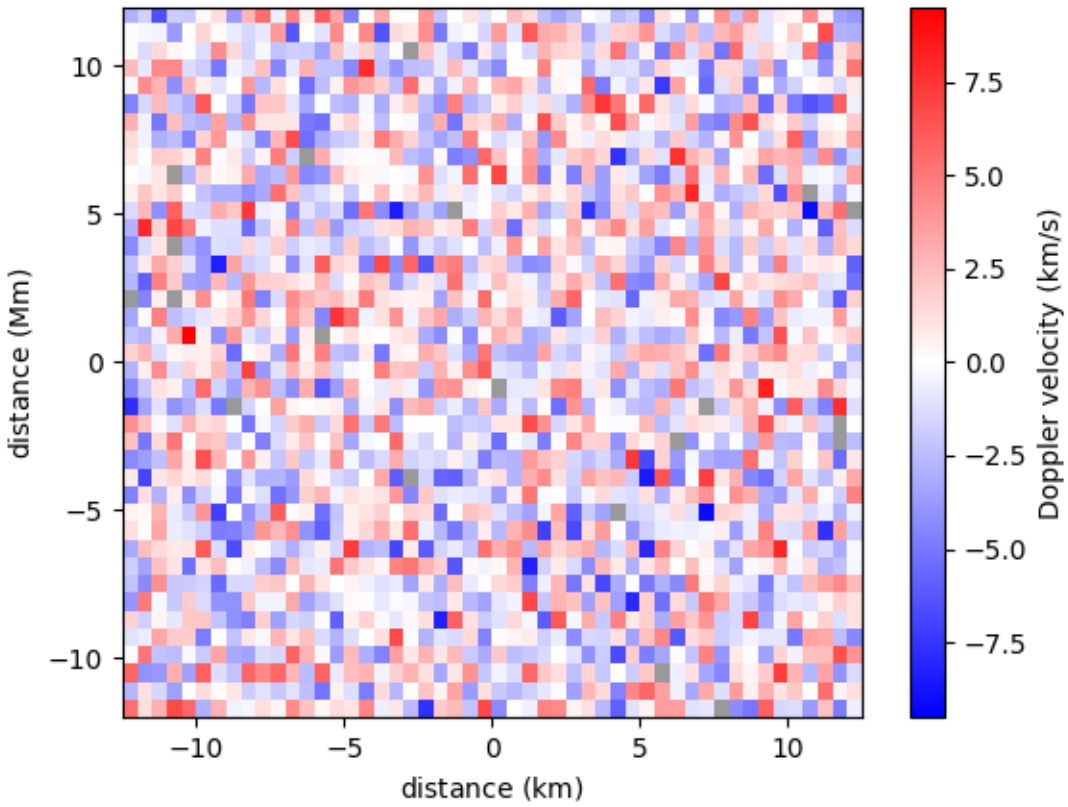
```
import astropy.units as u

plot_map(arr * 1000 * u.m / u.s)
```




```
<matplotlib.image.AxesImage object at 0x7fe936683a60>
```

A spatial resolution with units can be specified for each axis.

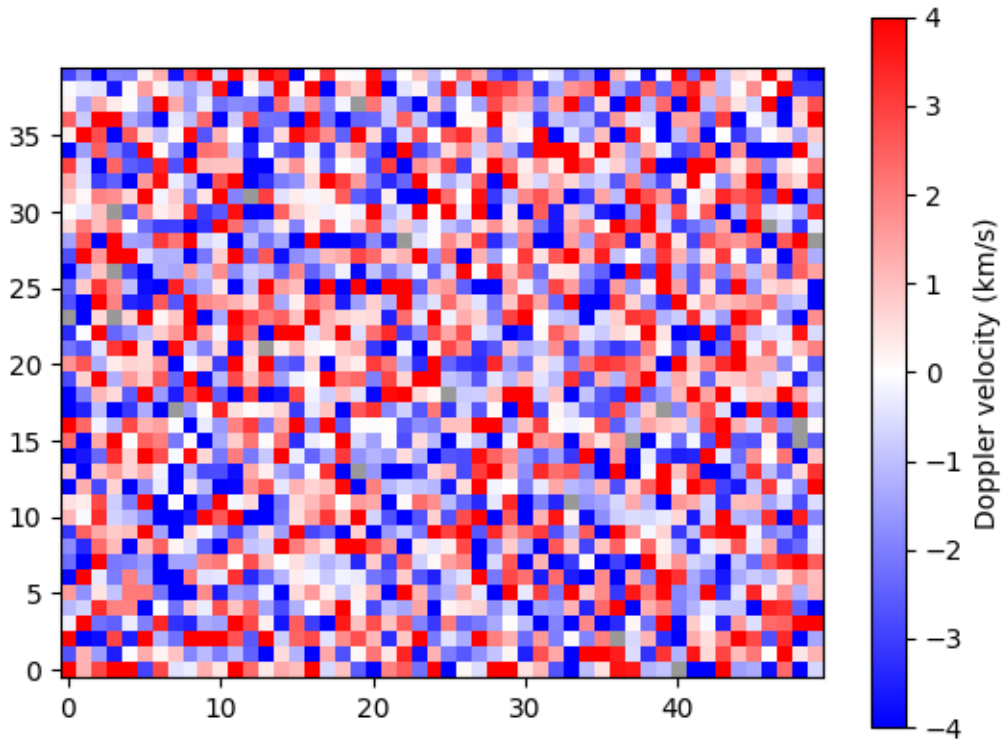
```
plot_map(arr, resolution=(0.5 * u.km, 0.6 * u.Mm), offset=(-25, -20))
```




```
<matplotlib.image.AxesImage object at 0x7fe9355a01f0>
```

A narrower range of velocities to be plotted can be requested with the `vmin` and `vmax` parameters. Classifications outside of the range will appear saturated. Providing only one of `vmin` and `vmax` with set the other such that zero is the midpoint.

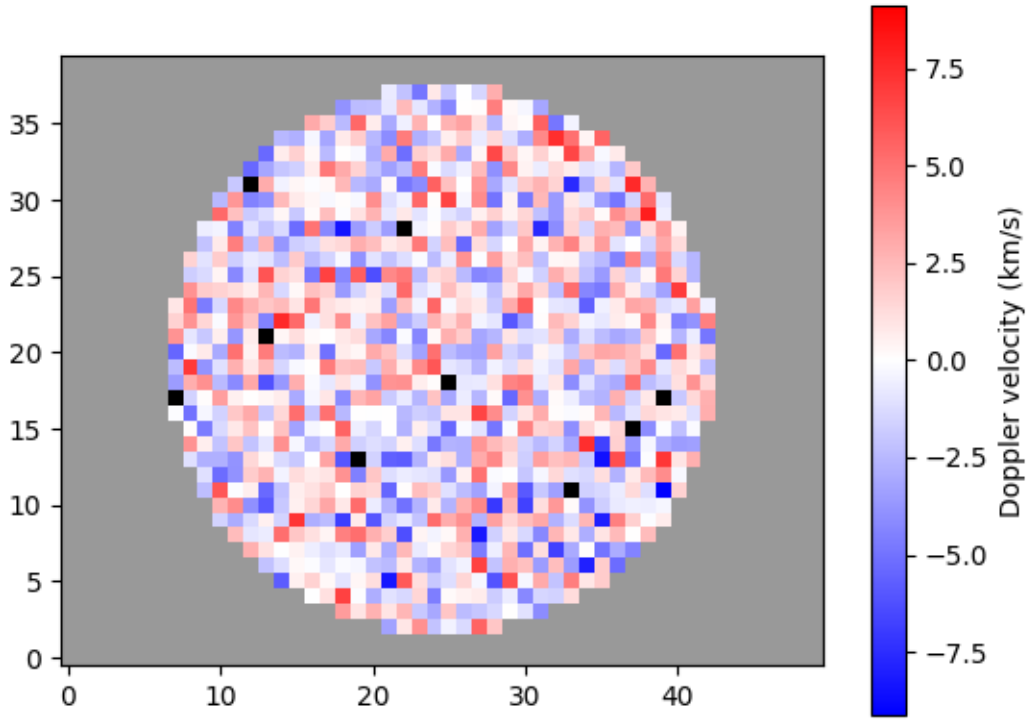
```
plot_map(arr, vmax=4)
```




```
<matplotlib.image.AxesImage object at 0x7fe9354a8a60>
```

A mask can be applied to the velocity array to isolate a region of interest. This functionally is useful if, for example, data only exist for a circular region and you want to distinguish between the pixels that are out of bounds and the data that were not successfully fitted.

```
from mcalf.utils.mask import genmask
mask = genmask(50, 40, 18)

plot_map(arr, ~mask)
```

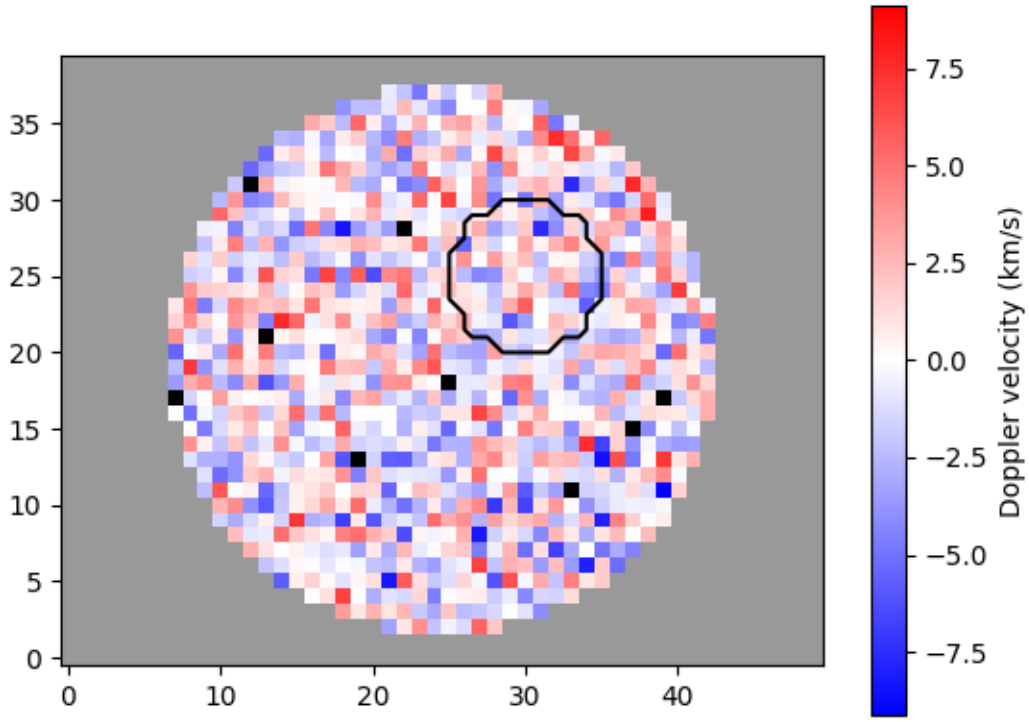



```
<matplotlib.image.AxesImage object at 0x7fe9364c7a60>
```

Notice how data out of bounds are grey, while data which were not fitted successfully are now black.

A region of interest, typically the umbra of a sunspot, can be outlined by passing a different mask.

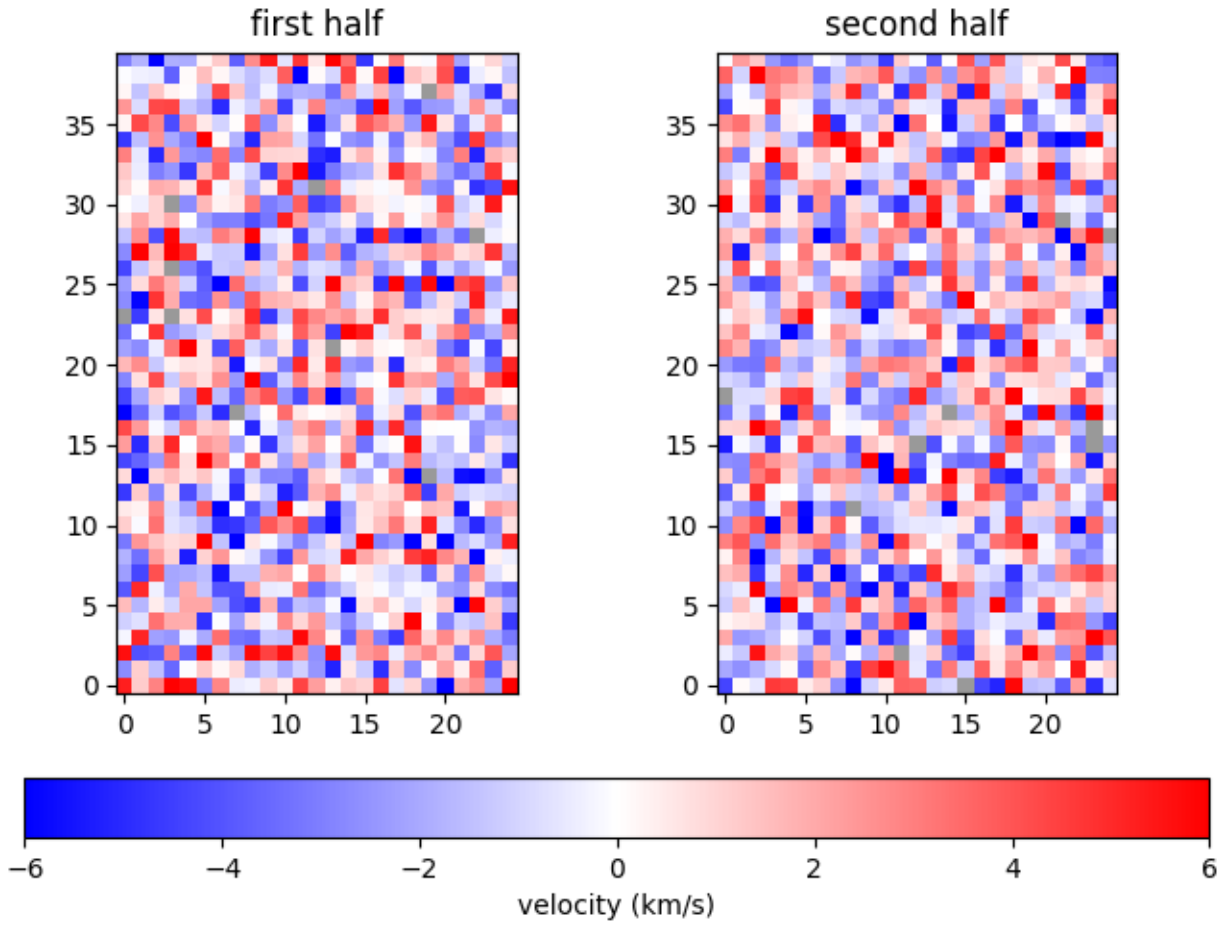
```
umbra_mask = genmask(50, 40, 5, 5, 5)

plot_map(arr, ~mask, umbra_mask)
```




```
<matplotlib.image.AxesImage object at 0x7fe935648610>
```

The plot_map function integrates well with matplotlib, allowing extensive flexibility.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, constrained_layout=True)

plot_map(arr[:, :25], vmax=6, ax=ax[0], show_colorbar=False)
im = plot_map(arr[:, 25:], vmax=6, ax=ax[1], show_colorbar=False)

fig.colorbar(im, ax=ax, location='bottom', label='velocity (km/s)')

ax[0].set_title('first half')
ax[1].set_title('second half')

plt.show()
```



Total running time of the script: (0 minutes 1.603 seconds)

Plot a spectrum

This is an example showing how to plot a spectrum with the *mcalf.visualisation.plot_spectrum()* function.

First we shall create a list of wavelengths, with a variable wavelength spacing. Next, we shall use the Voigt profile to generate spectral intensities at each of the wavelength points. Typically you would provide a spectrum obtained from observations.

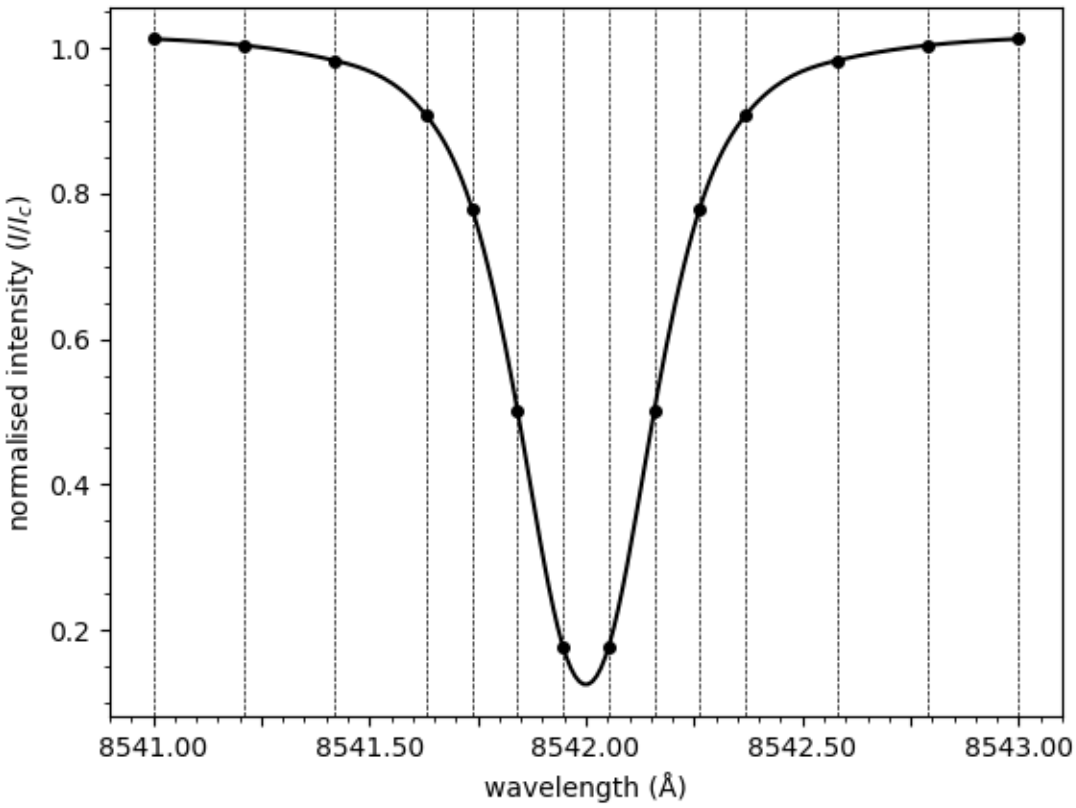
```
import numpy as np
wavelengths = np.linspace(8541, 8543, 20)
wavelengths = np.delete(wavelengths, np.s_[1:6:2])
wavelengths = np.delete(wavelengths, np.s_[-6::2])

from mcalf.profiles.voigt import voigt
spectrum = voigt(wavelengths, -526, 8542, 0.1, 0.1, 1242)
```

Next, we shall import *mcalf.visualisation.plot_spectrum()*.

```
from mcalf.visualisation import plot_spectrum
```

We can now simply plot the spectrum.

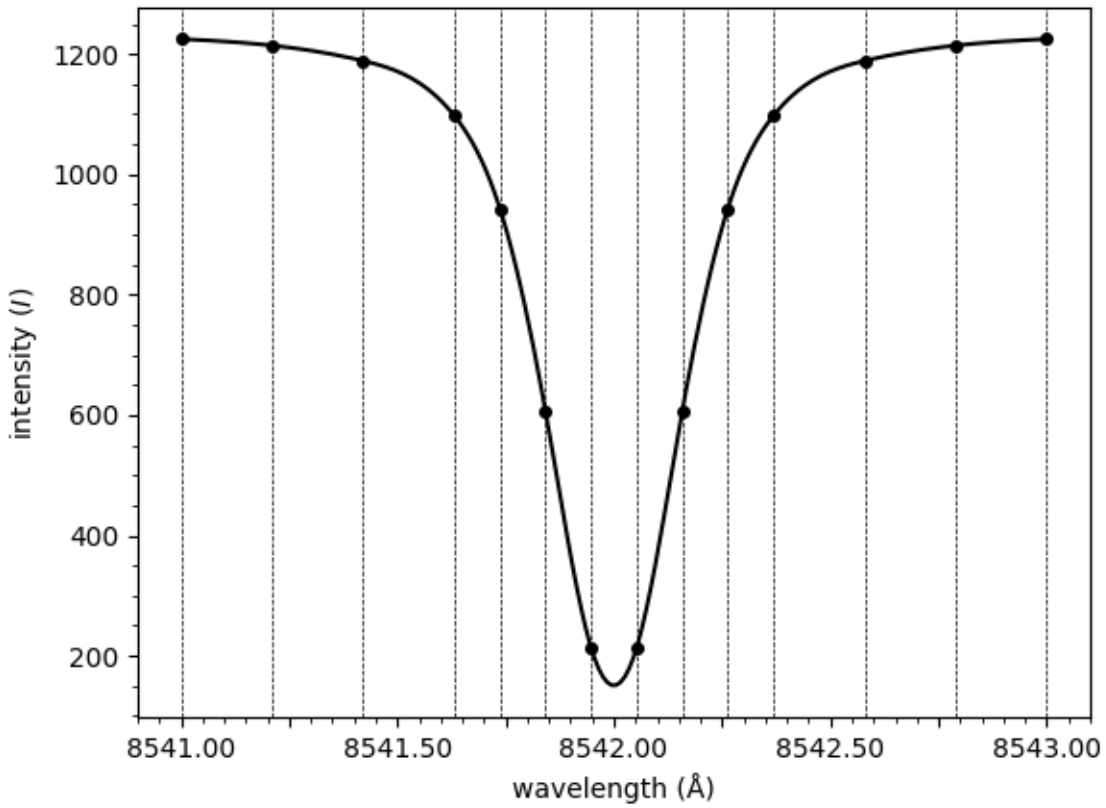
```
plot_spectrum(wavelengths, spectrum)
```




```
<Axes: xlabel='wavelength (Å)', ylabel='normalised intensity ($I/I_c$)'>
```

Notice how the spectrum above is normalised. The normalisation is applied by dividing through by the mean of the three rightmost points. To plot the raw spectrum,

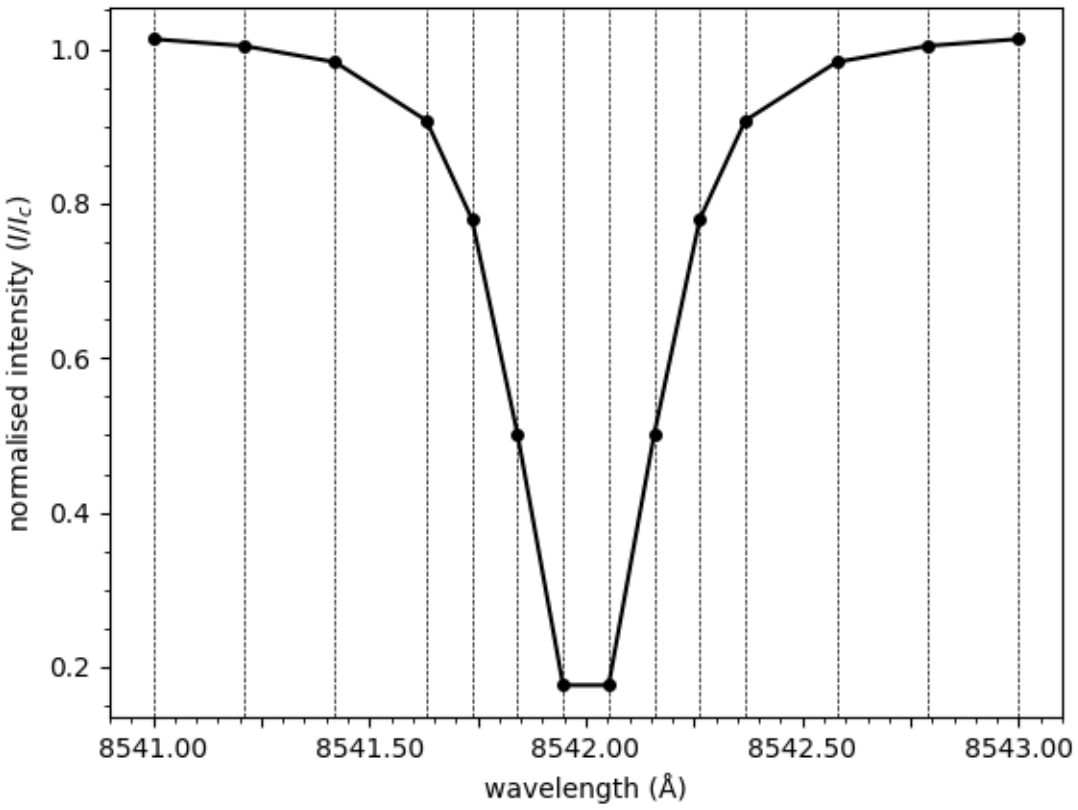
```
plot_spectrum(wavelengths, spectrum, normalised=False)
```




```
<Axes: xlabel='wavelength (Å)', ylabel='intensity ($I$)'>
```

The line connecting the points provided in the `spectrum` array above is smooth. This is due to spline interpolation being applied. Interpolation can be disabled, resulting in a straight line between each of the points.

```
plot_spectrum(wavelengths, spectrum, smooth=False)
```




```
<Axes: xlabel='wavelength (Å)', ylabel='normalised intensity ($I/I_c$)'>
```

Total running time of the script: (0 minutes 0.822 seconds)

example1: Basic usage of the package

`FittingIBIS.ipynb`

- View code
- `Download FittingIBIS.ipynb`

This file is an IPython Notebook containing examples of how to use the package to accomplish typical tasks.

FittingIBIS.pro

- Download FittingIBIS.pro

This file is similar to FittingIBIS.ipynb file, except it written is IDL. It is not recommended to use the IDL wrapper in production, just use it to explore the code if you are familiar with IDL and not Python. If you wish to use this package, please use the Python implementation. IDL is not fully supported in the current version of the code for reasons such as, the Python tuple datatype cannot be passed from IDL to Python, resulting in certain function calls not being possible.

config.yml

- Download config.yml

This is an example configuration file containing default parameters. This can be easier than setting the parameters in the code. The file follows the YAML format.

Labelling Tutorial

This Jupyter notebook provides a simple, semi-automated, method to produce a ground truth data set that can be used to train a neural network for use as a spectral shape classifier in the MCALF package. The following code can be adapted depending on the number of classifications that you want.

Download LabellingTutorial.ipynb

Load the required packages

[ ]:
```
import mcalf.models
from mcalf.utils.spec import normalise_spectrum
import requests
import numpy as np
from astropy.io import fits
import matplotlib.pyplot as plt
```

Download sample data

[ ]:
```
path = 'https://raw.githubusercontent.com/ConorMacBride/mcalf/main/examples/data/
↪ibis8542data/'

for file in ('wavelengths.txt', 'spectra.fits'):
    r = requests.get(path + file, allow_redirects=True)
    with open(file, 'wb') as f:
        f.write(r.content)
```

Load data files

[ ]:
```
wavelengths = np.loadtxt('wavelengths.txt')  # Original wavelengths

with fits.open('spectra.fits') as hdul:  # Raw spectral data
    datacube = np.asarray(hdul[0].data, dtype=np.float64)
```

Initialise the model that will use the labelled data

[ ]:
```
model = mcalf.models.IBIS8542Model(original_wavelengths=wavelengths)
```

Select the spectra to label

```
[ ]: n_points = 50

flat_choice = np.random.choice(np.arange(datacube[0].size), n_points, replace=False)
i_points, j_points = np.unravel_index(flat_choice, datacube[0].shape)
np.save('labelled_points.npy', np.array([i_points, j_points]))
```

```
[ ]: i_points, j_points = np.load('labelled_points.npy')
```

Select the spectra to label from the data file

```
[ ]: raw_spectra = datacube[:, i_points, j_points].T
```

Normalise each spectrum to be in range [0, 1]

```
[ ]: labelled_spectra = np.empty((len(raw_spectra), len(model.constant_wavelengths)))
for i in range(len(labelled_spectra)):
    labelled_spectra[i] = normalise_spectrum(raw_spectra[i], model=model)
```

Script to semi-automate the classification process

- Type a number 0 - 4 for assign a classification to the plotted spectrum
- Type 5 to skip and move on to the next spectrum
- Type `back` to move to the previous spectrum
- Type `exit` to give up (keeping ones already done)

The labels are present in the `labels` variable (-1 represents an unclassified spectrum)

```
[ ]: labels = np.full(len(labelled_spectra), -1, dtype=int)
i = 0
while i < len(labelled_spectra):

    # Show the spectrum to be classified along with description
    plt.figure(figsize=(15, 10))
    plt.plot(labelled_spectra[i])
    plt.show()
    print("i = {}".format(i))
    print("absorption --- both --- emission / skip")
    print("       0    1    2    3    4         5 ")

    # Ask for user's classification
    classification = input('Type [0-4]:')

    try:  # Must be an integer
        classification_int = int(classification)
    except ValueError:
        classification_int = -1  # Try current spectrum again

    if classification == 'back':
        i -= 1  # Go back to the previous spectrum
    elif classification == 'exit':
```

(continues on next page)

(continued from previous page)

```
        break  # Exit the loop, saving labels that were given
    elif 0 <= classification_int <= 4:  # Valid classification
        labels[i] = int(classification)  # Assign the classification to the spectrum
        i += 1  # Move on to the next spectrum
    elif classification_int == 5:
        i += 1  # Skip and move on to the next spectrum
    else:  # Invalid integer classification
        i += 0  # Try current spectrum again
```

Plot bar chart of classification populations

```
[ ]: unique, counts = np.unique(labels, return_counts=True)
     plt.figure()
     plt.bar(unique, counts)
     plt.title('Number of spectra in each classification')
     plt.xlabel('Classification')
     plt.ylabel('N_spectra')
     plt.show()
```

Overplot the spectra of each classification

```
[ ]: for classification in unique:
         plt.figure()
         for spectrum in labelled_spectra[labels == classification]:
             plt.plot(model.constant_wavelengths, spectrum)
         plt.title('Classification {}'.format(classification))
         plt.yticks([0, 1])
         plt.show()
```

Save the labelled spectra for use later

```
[ ]: np.save('labelled_data.npy', labelled_spectra)
     np.save('labels.npy', labels)
```

1.1.4 Contributing

Code of Conduct

If you find this package useful and have time to make it even better, you are very welcome to contribute to this package, regardless of how much prior experience you have. Types of ways you can contribute include, expanding the documentation with more use cases and examples, reporting bugs through the GitHub issue tracker, reviewing pull requests and the existing code, fixing bugs and implementing new features in the code. You are encouraged to submit any bug reports and pull requests directly to the GitHub repository.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms.

1.1.5 Citation

If you have used this package in work that leads to a publication, we would be very grateful if you could acknowledge your use of this package in the main text of the publication. Please cite the following publications,

> MacBride CD, Jess DB. 2021 MCALF: Multi-Component Atmospheric Line Fitting. *Journal of Open Source Software*. **6(61)**, 3265. (doi:10.21105/joss.03265)
>
> MacBride CD, Jess DB, Grant SDT, Khomenko E, Keys PH, Stangalini M. 2020 Accurately constraining velocity information from spectral imaging observations using machine learning techniques. *Philosophical Transactions of the Royal Society A*. **379**, 2190. (doi:10.1098/rsta.2020.0171)

Please also cite the Zenodo DOI for the package version you used. Please also consider integrating your code and examples into the package.

1.1.6 License

MCALF is licensed under the terms of the BSD 2-Clause license.

1.2 Code Reference

1.2.1 MCALF

mcalf Package

MCALF: Multi-Component Atmospheric Line Fitting

MCALF is an open-source Python package for accurately constraining velocity information from spectral imaging observations using machine learning techniques.

1.2.2 MCALF models

This sub-package contains:

- Base and sample models that can be adapted and fitted to any spectral imaging dataset.
- Models optimised for particular data sets that can be used directly.
- Data structures for storing and exporting the fitted parameters, as well as simplifying the calculation of velocities.

mcalf.models Package

Classes

FitResult(fitted_parameters, fit_info)	Class that holds the result of a fit.
FitResults(shape, n_parameters[, time])	Class that holds multiple fit results in a way that can be easily processed.
IBIS8542Model(*[, constant_wavelengths, ...])	Class for working with IBIS 8542 Å calcium II spectral imaging observations.
ModelBase(*[, original_wavelengths, ...])	Base class for spectral line model fitting.

FitResult

class mcalf.models.**FitResult**(*fitted_parameters*, *fit_info*)

Bases: object

Class that holds the result of a fit.

Parameters

- **fitted_parameters** (*numpy.ndarray*) – The parameters fitted.
- **fit_info** (*dict*) – Additional information on the fit including at least 'classification', 'profile', 'success', 'chi2' and 'index'.

parameters

The parameters fitted.

Type
numpy.ndarray

classification

Classification of the fitted spectrum.

Type
int

profile

Profile of the fitted spectrum.

Type
str

success

Whether the fit was completed successfully.

Type
bool

chi2

Chi-squared value for the fit.

Type
float

index

Index ([<time>, <row>, <column>]) of the spectrum in the spectral array.

Type
list

__dict__

Other attributes may be present depending on the *fit_info* used.

Methods Summary

plot(model, **kwargs)	Plot the data and fitted parameters.
velocity(model[, vtype])	Calculate the Doppler velocity of the fit using *model* parameters.

Methods Documentation

plot(*model*, ***kwargs*)

Plot the data and fitted parameters.

This calls the *plot* method on *model* but will plot for this FitResult object. See the model's *plot* method for more details.

Parameters

- **model** (child class of *ModelBase*) – The model object to plot with.
- ****kwargs** – See the *model.plot* method for more details.

velocity(*model*, *vtype='quiescent'*)

Calculate the Doppler velocity of the fit using *model* parameters.

Parameters

- **model** (child class of *ModelBase*) – The model object to take parameters from.
- **vtype** (*{'quiescent', 'active'}, default='quiescent'*) – The velocity type to find.

Returns
velocity – The calculated velocity.

Return type
float

FitResults

class mcalf.models.**FitResults**(*shape*, *n_parameters*, *time=None*)

Bases: object

Class that holds multiple fit results in a way that can be easily processed.

Parameters

- **shape** (*tuple of int*) – The number of rows and columns to hold data for, e.g. (n_rows, n_columns).
- **n_parameters** (*int*) – The number of fitted parameters per spectrum that need to be stored.
- **time** (*int, optional, default=None*) – The time the *FitResults* object will store data for. Optional, but if it is set, only *FitResult* objects with a matching time can be appended.

parameters

Array of fitted parameters.

Type
numpy.ndarray, shape=(row, column, parameter)

classifications

Array of classifications.

Type
numpy.ndarray of int, shape=(row, column)

profile

Array of profiles.

Type
numpy.ndarray of str, shape=(row, column)

success

Array of success statuses.

Type
numpy.ndarray of bool, shape=(row, column)

chi2

Array of chi-squared values.

Type
numpy.ndarray, shape=(row, column)

time

Time index that the *FitResult* object refers to (if provided).

Type
int, default=None

n_parameters

Number of parameters in the last dimension of *parameters*.

Type
int

Methods Summary

append(result)	Append a *FitResult* object to the *FitResults* object.
save(filename[, model])	Saves the FitResults object to a FITS file.
velocities(model[, row, column, vtype])	Calculate the Doppler velocities of the fit results using *model* parameters.

Methods Documentation

append(*result*)

Append a *FitResult* object to the *FitResults* object.

Parameters
result (FitResult) – *FitResult* object to append.

save(*filename*, *model=None*)

Saves the FitResults object to a FITS file.

Parameters

- **filename** (*file path, file object or file-like object*) – FITS file to write to. If a file object, must be opened in a writeable mode.
- **model** (*child class of* mcalf.models.ModelBase*, optional, default=None*) – If provided, use this model to calculate and include both quiescent and active Doppler velocities. The stationary line core value will also be added to the *SLC* card in the primary HDU header.

Notes

Saves a FITS file to the location specified by *filename*. All the parameters are stored in a separate, named, HDU.

velocities(*model*, *row=None*, *column=None*, *vtype='quiescent'*)

Calculate the Doppler velocities of the fit results using *model* parameters.

Parameters

- **model** (*child class of* mcalf.models.ModelBase) – The model object to take parameters from.
- **row** (*int, list, array_like, iterable, optional, default=None*) – The row indices to find velocities for. All if omitted.
- **column** (*int, list, array_like, iterable, optional, default=None*) – The column indices to find velocities for. All if omitted.
- **vtype** (*{'quiescent', 'active'}, default='quiescent'*) – The velocity type to find.

Returns

velocities – The calculated velocities for the specified *row* and *column* positions.

Return type

numpy.ndarray, shape=(row, column)

IBIS8542Model

class mcalf.models.**IBIS8542Model**(**, constant_wavelengths=None, delta_lambda=0.05, sigma=None, prefilter_response=None, prefilter_ref_main=None, prefilter_ref_wvscl=None, output=None, config=None, stationary_line_core='8542.099145376844', absorption_guess=[-1000, '8542.099145376844', 0.2, 0.1], emission_guess=[1000, '8542.099145376844', 0.2, 0.1], absorption_min_bound=[-inf, 'stationary_line_core-0.15', 1e-06, 1e-06], emission_min_bound=[0, -inf, 1e-06, 1e-06], absorption_max_bound=[0, 'stationary_line_core+0.15', 1, 1], emission_max_bound=[inf, inf, 1, 1], absorption_x_scale=[1500, 0.2, 0.3, 0.5], emission_x_scale=[1500, 0.2, 0.3, 0.5], random_state=None, impl=<function voigt_faddeeva>*)

Bases: *ModelBase*

Class for working with IBIS 8542 Å calcium II spectral imaging observations.

Parameters

- **absorption_guess** (*array_like, length=4, optional, default=[-1000, stationary_line_core, 0.2, 0.1]*) – Initial guess to take when fitting the absorption Voigt profile.

- **emission_guess** (*array_like, length=4, optional, default=[1000, stationary_line_core, 0.2, 0.1]*) – Initial guess to take when fitting the emission Voigt profile.
- **absorption_min_bound** (*array_like, length=4, optional, default=[-np.inf, stationary_line_core-0.15, 1e-6, 1e-6]*) – Minimum bounds for all the absorption Voigt profile parameters in order of the function's arguments.
- **emission_min_bound** (*array_like, length=4, optional, default=[0, -np.inf, 1e-6, 1e-6]*) – Minimum bounds for all the emission Voigt profile parameters in order of the function's arguments.
- **absorption_max_bound** (*array_like, length=4, optional, default=[0, stationary_line_core+0.15, 1, 1]*) – Maximum bounds for all the absorption Voigt profile parameters in order of the function's arguments.
- **emission_max_bound** (*array_like, length=4, optional, default=[np.inf, np.inf, 1, 1]*) – Maximum bounds for all the emission Voigt profile parameters in order of the function's arguments.
- **absorption_x_scale** (*array_like, length=4, optional, default=[1500, 0.2, 0.3, 0.5]*) – Characteristic scale for all the absorption Voigt profile parameters in order of the function's arguments.
- **emission_x_scale** (*array_like, length=4, optional, default=[1500, 0.2, 0.3, 0.5]*) – Characteristic scale for all the emission Voigt profile parameters in order of the function's arguments.
- **random_state** (*int, numpy.random.RandomState, optional, default=None*) – Determines random number generation for weights and bias initialisation of the default *neural_network*. Pass an int for reproducible results across multiple function calls.
- **impl** (*callable, optional, default=voigt_faddeeva*) – Voigt implementation to use.
- **original_wavelengths** (*array_like*) – One-dimensional array of wavelengths that correspond to the uncorrected spectral data.
- **stationary_line_core** (*float, optional, default=8542.099145376844*) – Wavelength of the stationary line core.
- **constant_wavelengths** (*array_like, ndim=1, optional, default= see description*) – The desired set of wavelengths that the spectral data should be rescaled to represent. It is assumed that these have constant spacing, but that may not be a requirement if you specify your own array. The default value is an array from the minimum to the maximum wavelength of *original_wavelengths* in constant steps of *delta_lambda*, overshooting the upper bound if the maximum wavelength has not been reached.
- **delta_lambda** (*float, optional, default=0.05*) – The step used between each value of *constant_wavelengths* when its default value has to be calculated.
- **sigma** (*list of array_like or bool, length=(2, n_wavelengths), optional, default=[type1, type2]*) – A list of different sigma that are used to weight particular wavelengths along the spectra when fitting. The fitting method will expect to be able to choose a sigma array from this list at a specific index. It's default value is *[generate_sigma(i, constant_wavelengths, stationary_line_core) for i in [1, 2]]*. See *mcalf.utils.spec.generate_sigma()* for more information. If bool, True will generate the default sigma value regardless of the value specified in *config*, and False will set *sigma* to be all ones, effectively disabling it.

- **prefilter_response** (*array_like, length=n_wavelengths, optional, default= see note*) – Each constant wavelength scaled spectrum will be corrected by dividing it by this array. If *prefilter_response* is not given, and *prefilter_ref_main* and *prefilter_ref_wvscl* are not given, *prefilter_response* will have a default value of *None*.
- **prefilter_ref_main** (*array_like, optional, default= None*) – If *prefilter_response* is not specified, this will be used along with *prefilter_ref_wvscl* to generate the default value of *prefilter_response*.
- **prefilter_ref_wvscl** (*array_like, optional, default=None*) – If *prefilter_response* is not specified, this will be used along with *prefilter_ref_main* to generate the default value of *prefilter_response*.
- **config** (*str, optional, default=None*) – Filename of a *.yml* file (relative to current directory) containing the initialising parameters for this object. Parameters provided explicitly to the object upon initialisation will override any provided in this file. All (or some) parameters that this object accepts can be specified in this file, except *neural_network* and *config*. Each line of the file should specify a different parameter and be formatted like *emission_guess: '[-inf, wl-0.15, 1e-6, 1e-6]'* or *original_wavelengths: 'original.fits'* for example. When specifying a string, use 'inf' to represent *np.inf* and 'wl' to represent *stationary_line_core* as shown. If the string matches a file, *mcalf.utils.misc.load_parameter()* is used to load the contents of the file.
- **output** (*str, optional, default=None*) – If the program wants to output data, it will place it relative to the location specified by this parameter. Some methods will only save data to a file if this parameter is not *None*. Such cases will be documented where relevant.

absorption_guess

Initial guess to take when fitting the absorption Voigt profile.

Type
array_like, length=4, optional, default=[-1000, stationary_line_core, 0.2, 0.1]

emission_guess

Initial guess to take when fitting the emission Voigt profile.

Type
array_like, length=4, optional, default=[1000, stationary_line_core, 0.2, 0.1]

absorption_min_bound

Minimum bounds for all the absorption Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[-np.inf, stationary_line_core-0.15, 1e-6, 1e-6]

emission_min_bound

Minimum bounds for all the emission Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[0, -np.inf, 1e-6, 1e-6]

absorption_max_bound

Maximum bounds for all the absorption Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[0, stationary_line_core+0.15, 1, 1]

emission_max_bound

Maximum bounds for all the emission Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[np.inf, np.inf, 1, 1]

`absorption_x_scale`

Characteristic scale for all the absorption Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[1500, 0.2, 0.3, 0.5]

`emission_x_scale`

Characteristic scale for all the emission Voigt profile parameters in order of the function's arguments.

Type
array_like, length=4, optional, default=[1500, 0.2, 0.3, 0.5]

`random_state`

Determines random number generation for weights and bias initialisation of the default *neural_network*. Pass an int for reproducible results across multiple function calls.

Type
int, numpy.random.RandomState, optional, default=None

`impl`

Voigt implementation to use.

Type
callable, optional, default=voigt_faddeeva

`quiescent_wavelength`

The index within the fitted parameters of the absorption Voigt line core wavelength.

Type
int, default=1

`active_wavelength`

The index within the fitted parameters of the emission Voigt line core wavelength.

Type
int, default=5

`original_wavelengths`

One-dimensional array of wavelengths that correspond to the uncorrected spectral data.

Type
array_like

`stationary_line_core`

Wavelength of the stationary line core.

Type
float, optional, default=8542.099145376844

`neural_network`

The `sklearn.neural_network.MLPClassifier` object (or similar) that will be used to classify the spectra. Defaults to a `sklearn.model_selection.GridSearchCV` with `MLPClassifier(solver='lbfgs', hidden_layer_sizes=(40,), max_iter=1000)` for best *alpha* selected from *[1e-5, 2e-5, 3e-5, 4e-5, 5e-5, 6e-5, 7e-5, 8e-5, 9e-5]*.

Type
sklearn.neural_network.MLPClassifier, optional, default= see description

`constant_wavelengths`

The desired set of wavelengths that the spectral data should be rescaled to represent. It is assumed that these have constant spacing, but that may not be a requirement if you specify your own array. The default value is an array from the minimum to the maximum wavelength of *original_wavelengths* in constant steps of *delta_lambda*, overshooting the upper bound if the maximum wavelength has not been reached.

Type
array_like, ndim=1, optional, default= see description

`sigma`

A list of different sigma that are used to weight particular wavelengths along the spectra when fitting. The fitting method will expect to be able to choose a sigma array from this list at a specific index. It's default value is *[generate_sigma(i, constant_wavelengths, stationary_line_core) for i in [1, 2]]*. See `mcalf.utils.spec.generate_sigma()` for more information. If bool, True will generate the default sigma value regardless of the value specified in *config*, and False will set *sigma* to be all ones, effectively disabling it.

Type
list of array_like or bool, length=(2, n_wavelengths), optional, default=[type1, type2]

`prefilter_response`

Each constant wavelength scaled spectrum will be corrected by dividing it by this array. If *prefilter_response* is not given, and *prefilter_ref_main* and *prefilter_ref_wvscl* are not given, *prefilter_response* will have a default value of *None*.

Type
array_like, length=n_wavelengths, optional, default= see note

`output`

If the program wants to output data, it will place it relative to the location specified by this parameter. Some methods will only save data to a file if this parameter is not *None*. Such cases will be documented where relevant.

Type
str, optional, default=None

`array`

Array holding spectra.

Type
numpy.ndarray, dimensions are ['time', 'row', 'column', 'spectra']

`background`

Array holding spectral backgrounds.

Type
numpy.ndarray, dimensions are ['time', 'row', 'column']

Attributes Summary

default_ibis8542model_kwargs	
default_kwargs	
default_modelbase_kwargs	
stationary_line_core	

Methods Summary

classify_spectra([time, row, column, ...])	Classify the specified spectra.
fit([time, row, column, spectrum, ...])	Fits the model to specified spectra.
fit_spectrum(spectrum, **kwargs)	Fits the specified spectrum array.
get_spectra([time, row, column, spectrum, ...])	Gets corrected spectra from the spectral array.
load_array(array[, names])	Load an array of spectra.
load_background(array[, names])	Load an array of spectral backgrounds.
plot([fit, time, row, column, spectrum, ...])	Plots the data and fitted parameters.
plot_separate(*args, **kwargs)	Plot the fitted profiles separately.
plot_subtraction(*args, **kwargs)	Plot the spectrum with the emission fit subtracted from it.
test(X, y)	Test the accuracy of the trained neural network.
train(X, y)	Fit the neural network model to spectra matrix X and spectra labels y.

Attributes Documentation

default_ibis8542model_kwargs = {'absorption_guess': [-1000, '8542.099145376844', 0.2, 0.1], 'absorption_max_bound': [0, 'stationary_line_core+0.15', 1, 1], 'absorption_min_bound': [-inf, 'stationary_line_core-0.15', 1e-06, 1e-06], 'absorption_x_scale': [1500, 0.2, 0.3, 0.5], 'emission_guess': [1000, '8542.099145376844', 0.2, 0.1], 'emission_max_bound': [inf, inf, 1, 1], 'emission_min_bound': [0, -inf, 1e-06, 1e-06], 'emission_x_scale': [1500, 0.2, 0.3, 0.5], 'impl': <function voigt_faddeeva>, 'random_state': None, 'stationary_line_core': '8542.099145376844'}

default_kwargs = {'absorption_guess': [-1000, '8542.099145376844', 0.2, 0.1], 'absorption_max_bound': [0, 'stationary_line_core+0.15', 1, 1], 'absorption_min_bound': [-inf, 'stationary_line_core-0.15', 1e-06, 1e-06], 'absorption_x_scale': [1500, 0.2, 0.3, 0.5], 'emission_guess': [1000, '8542.099145376844', 0.2, 0.1], 'emission_max_bound': [inf, inf, 1, 1], 'emission_min_bound': [0, -inf, 1e-06, 1e-06], 'emission_x_scale': [1500, 0.2, 0.3, 0.5], 'impl': <function voigt_faddeeva>, 'random_state': None, 'stationary_line_core': '8542.099145376844'}

default_modelbase_kwargs = {'constant_wavelengths': None, 'delta_lambda': 0.05, 'original_wavelengths': None, 'output': None, 'prefilter_ref_main': None, 'prefilter_ref_wvscl': None, 'prefilter_response': None, 'sigma': None, 'stationary_line_core': None}

stationary_line_core

Methods Documentation

classify_spectra(*time=None*, *row=None*, *column=None*, *spectra=None*, *only_normalise=False*)

Classify the specified spectra.

Will also normalise each spectrum such that its intensity will range from zero to one.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, a numpy.ndarray, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectra** (*numpy.ndarray, optional, default=None*) – The explicit spectra to classify. If *only_normalise* is False, this must be 1D. However, if *only_normalise* is set to true, *spectra* can be of any dimension. It is assumed that the final dimension is wavelengths, so return shape will be the same as *spectra*, except with no final wavelengths dimension.
- **only_normalise** (*bool, optional, default=False*) – Whether the single spectrum given in *spectra* should not be interpolated and corrected. If set to true, the only processing applied to *spectra* will be a normalisation to be in range 0 to 1.

Returns

classifications – Array of classifications with the same time, row and column indices as *spectra*.

Return type

numpy.ndarray

See also:

train
Train the neural network.

test
Test the accuracy of the neural network.

get_spectra
Get processed spectra from the objects *array* attribute.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8542.1, 8542.2, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load a trained neural network:

```
>>> import pickle
>>> pkl = open('trained_neural_network.pkl', 'rb')
>>> model.neural_network = pickle.load(pkl)
```

Classify an individual spectrum:

```
>>> spectrum = np.random.rand(30)
>>> model.classify_spectra(spectra=spectrum)
array([2])
```

When `only_normalise=True`, classify an n-dimensional spectral array:

```
>>> spectra = np.random.rand(5, 4, 3, 2, 30)
>>> model.classify_spectra(spectra=spectra, only_normalise=True).shape
(5, 4, 3, 2)
```

Load spectra from a file and classify:

```
>>> from astropy.io import fits
>>> spectra = fits.open('spectra_0000.fits')[0].data
>>> model.load_array(spectra, names=['wavelength', 'column', 'row'])
>>> model.classify_spectra(column=range(10, 15), row=[7, 16])
array([[[0, 2, 0, 3, 0],
        [4, 0, 1, 0, 0]]])
```

fit(*time=None*, *row=None*, *column=None*, *spectrum=None*, *classifications=None*, *background=None*, *n_pools=None*, ***kwargs*)

Fits the model to specified spectra.

Fits the model to an array of spectra using multiprocessing if requested.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, `numpy.ndarray`, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectrum** (*numpy.ndarray, ndim=1, optional, default=None*) – The explicit spectrum to fit the model to.

- **classifications** (*int or array_like, optional, default=None*) – Classifications to determine the fitted profile to use. Will use neural network to classify them if not. If a multidimensional array, must have the same shape as [*time*, *row*, *column*]. Dimensions that would have length of 1 can be excluded.
- **background** (*float, optional, default=None*) – If provided, this value will be subtracted from the explicit spectrum provided in *spectrum*. Will not be applied to spectra found from the indices, use the *load_background()* method instead.
- **n_pools** (*int, optional, default=None*) – The number of processing pools to calculate the fitting over. This allocates the fitting of different spectra to *n_pools* separate worker processes. When processing a large number of spectra this will make the fitting process take less time overall. It also distributes such that each worker process has the same ratio of classifications to process. This should balance out the workload between workers. If few spectra are being fitted, performance may decrease due to the overhead associated with splitting the evaluation over separate processes. If *n_pools* is not an integer greater than zero, it will fit the spectrum with a for loop.
- ****kwargs** – Extra keyword arguments to pass to *_fit()*.

Returns

result – Outcome of the fits returned as a list of *FitResult* objects.

Return type

list of *FitResult*, length=n_spectra

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Set up the neural network classifier:

```
>>> model.neural_network = ...  # load an untrained classifier
>>> model.train(...)
>>> model.test(...)
```

Load the spectra and background array:

```
>>> model.load_array(...)
>>> model.load_background(...)
```

Fit a subset of the loaded spectra, using 5 processing pools:

```
>>> fits = model.fit(row=range(3, 5), column=range(200), n_pools=5)
>>> fits
['Successful FitResult with ________ profile of classification 0',
 'Successful FitResult with ________ profile of classification 2',
 ...
 'Successful FitResult with ________ profile of classification 0',
 'Successful FitResult with ________ profile of classification 4']
```

Merge the fit results into a *FitResults* object:

```
>>> results = mcalf.models.FitResults((500, 500), 8)
>>> for fit in fits:
...     results.append(fit)
```

See *fit_spectrum()* examples for how to manually providing a *spectrum* to fit.

fit_spectrum(*spectrum*, ***kwargs*)

Fits the specified spectrum array.

Passes the spectrum argument to the *fit()* method. For easily iterating over a list of spectra.

Parameters

- **spectrum** (*numpy.ndarray, ndim=1*) – The explicit spectrum.
- ****kwargs** – Extra keyword arguments to pass to *fit()*.

Returns

result – Result of the fit.

Return type

FitResult

See also:

fit

General fitting method.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Quickly provide a spectrum and fit it. Remember that the model must be optimised for the spectra that it is asked to fit. In this example the neural network is not called upon to classify the provided spectrum as a classification is provided directly:

```
>>> spectrum = np.random.rand(30)
>>> model.fit_spectrum(spectrum, classifications=0, background=142.2)
Successful FitResult with ________ profile of classification 0
```

As the spectrum is provided manually, any background value must also be provided manually. Alternatively, the background can be subtracted before passing to the function, as by default, no background is subtracted:

```
>>> model.fit_spectrum(spectrum - 142.2, classifications=0)
Successful FitResult with ________ profile of classification 0
```

get_spectra(*time=None*, *row=None*, *column=None*, *spectrum=None*, *correct=True*, *background=False*)

Gets corrected spectra from the spectral array.

Takes either a set of indices or an explicit spectrum and optionally applied corrections and background removal.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, a `numpy.ndarray`, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectrum** (*ndarray of ndim=1, optional, default=None*) – The explicit spectrum. If provided, *time*, *row*, and *column* are ignored.
- **correct** (*bool, optional, default=True*) – Whether to reinterpolate the spectrum and apply the prefilter correction (if exists).
- **background** (*bool, optional, default=False*) – Whether to include the background in the outputted spectra. Only removes the background if the relevant background array has been loaded. Does not remove background is processing an explicit spectrum.

Returns
spectra

Return type
ndarray

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Provide a single spectrum for processing, and notice output is 1D:

```
>>> spectrum = model.get_spectra(spectrum=np.random.rand(30))
>>> spectrum.ndim
1
```

Load an array of spectra:

```
>>> spectra = np.random.rand(3, 4, 30)
>>> model.load_array(spectra, names=['column', 'row', 'wavelength'])
```

Extract a single (unprocessed) spectrum from the loaded array, and notice output is 4D:

```
>>> spectrum = model.get_spectra(row=1, column=0, correct=False)
>>> spectrum.shape
(1, 1, 1, 30)
>>> (spectrum[0, 0, 0] == spectra[0, 1]).all()
True
```

Extract an array of spectra, and notice output is 4D, and with dimensions time, row, column, wavelength regardless of the original dimensions and order:

```
>>> spectrum = model.get_spectra(row=range(4), column=range(3))
>>> spectrum.shape
(1, 4, 3, 30)
```

Notice that the time index can be excluded, as the loaded array only represents a single time. However, in this case leaving out *row* or *column* results in an error as it is ambiguous:

```
>>> spectrum = model.get_spectra(row=range(4))
Traceback (most recent call last):
 ...
ValueError: column index must be specified as multiple indices exist
```

load_array(*array*, *names=None*)

Load an array of spectra.

Load *array* with dimension names *names* into the *array* parameter of the model object.

Parameters

- **array** (*numpy.ndarray, ndim>1*) – An array containing at least two spectra.
- **names** (*list of str, length=`array.ndim`*) – List of dimension names for *array*. Valid dimension names are 'time', 'row', 'column' and 'wavelength'. 'wavelength' is a required dimension.

See also:

load_background
Load an array of spectral backgrounds.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> from astropy.io import fits
>>> wavelengths = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0]
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load spectra from a file:

```
>>> spectra = fits.open('spectra_0000.fits')[0].data
>>> model.load_array(spectra, names=['wavelength', 'column', 'row'])
```

load_background(*array*, *names=None*)

Load an array of spectral backgrounds.

Load *array* with dimension names *names* into *background* parameter of the model object.

Parameters

- **array** (*numpy.ndarray, ndim>0*) – An array containing at least two backgrounds.
- **names** (*list of str, length=`array.ndim`*) – List of dimension names for *array*. Valid dimension names are 'time', 'row' and 'column'.

See also:

load_array
Load and array of spectra.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> from astropy.io import fits
>>> wavelengths = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0]
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load background array from a file:

```
>>> background = fits.open('background_0000.fits')[0].data
>>> model.load_background(background, names=['column', 'row'])
```

plot(*fit=None*, *time=None*, *row=None*, *column=None*, *spectrum=None*, *classification=None*, *background=None*, *sigma=None*, *stationary_line_core=None*, ***kwargs*)

Plots the data and fitted parameters.

Parameters

- **fit** (*mcalf.models.FitResult or list or array_like, optional, default=None*) – The fitted parameters to plot with the data. Can extract the necessary plot metadata from the fit object. Otherwise, *fit* should be the parameters to be fitted to either a Voigt or double Voigt profile depending on the number of parameters fitted.
- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, `numpy.ndarray`, a Python range, etc. can be used. If not provided, will be taken from *fit* if it is a *FitResult* object, unless a *spectrum* is provided.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectrum** (*numpy.ndarray, length=`original_wavelengths`, ndim=1, optional, default=None*) – The explicit spectrum to plot along with a fit (if specified).
- **classification** (*int, optional, default=None*) – Used to determine which sigma profile to use. See `_get_sigma()` for more details. If not provided, will be taken from *fit* if it is a *FitResult* object, unless a *spectrum* is provided.
- **background** (*float or array_like, length=n_constant_wavelengths, optional, default= see note*) – Background to added to the fitted profiles. If a *spectrum* is given, this will default to zero, otherwise the value loaded by *load_background()* will be used.
- **sigma** (*int or array_like, optional, default=None*) – Explicit sigma index or profile. See `_get_sigma()` for details.
- **stationary_line_core** (*float, optional, default=`stationary_line_core`*) – The stationary line core wavelength to mark on the plot.

- ****kwargs** – Other parameters used to adjust the plotting. See *mcalf.visualisation.plot_ibis8542()* for full details.
  - *separate* – See *plot_separate()*.
  - *subtraction* – See *plot_subtraction()*.
  - *sigma_scale* – A factor to multiply the error bars to change their prominence.

See also:

plot_separate
Plot the fit parameters separately.

plot_subtraction
Plot the spectrum with the emission fit subtracted from it.

mcalf.models.FitResult.plot
Plotting method on the fit result.

Examples

- *Plot a fitted spectrum*

plot_separate(**args*, ***kwargs*)

Plot the fitted profiles separately.

If multiple profiles exist, fit them separately. Arguments are the same as the *plot()* method.

See also:

plot
General plotting method.

plot_subtraction
Plot the spectrum with the emission fit subtracted from it.

mcalf.models.FitResult.plot
Plotting method on the fit result.

plot_subtraction(**args*, ***kwargs*)

Plot the spectrum with the emission fit subtracted from it.

If multiple profiles exist, subtract the fitted emission from the raw data. Arguments are the same as the *plot()* method.

See also:

plot
General plotting method.

plot_separate
Plot the fit parameters separately.

mcalf.models.FitResult.plot
Plotting method on the fit result.

test(*X*, *y*)

Test the accuracy of the trained neural network.

Prints a table of results showing:

1) the percentage of predictions that equal the target labels;
2) the average classification deviation and standard deviation from the ground truth classification for each labelled classification;
3) the average classification deviation and standard deviation overall.

If the model object has an output parameter, it will create a CSV file (output/neural_network/test.csv) listing the predictions and ground truth data.

Parameters

- **X** (*numpy.ndarray or sparse matrix, shape=(n_spectra, n_wavelengths)*) – The input spectra.
- **y** (*numpy.ndarray, shape= (n_spectra,) or (n_spectra, n_outputs)*) – The target class labels.

See also:

train
Train the neural network.

train(*X*, *y*)

Fit the neural network model to spectra matrix X and spectra labels y.

Calls the *fit()* method on the *neural_network* parameter of the model object.

Parameters

- **X** (*numpy.ndarray or sparse matrix, shape=(n_spectra, n_wavelengths)*) – The input spectra.
- **y** (*numpy.ndarray, shape= (n_spectra,) or (n_spectra, n_outputs)*) – The target class labels.

See also:

test
Test how well the neural network has been trained.

ModelBase

class mcalf.models.**ModelBase**(*, *original_wavelengths=None*, *constant_wavelengths=None*, *delta_lambda=0.05*, *sigma=None*, *prefilter_response=None*, *prefilter_ref_main=None*, *prefilter_ref_wvscl=None*, *output=None*, *config=None*)

Bases: object

Base class for spectral line model fitting.

Warning: This class should not be used directly. Use derived classes instead.

Parameters

- **original_wavelengths** (*array_like*) – One-dimensional array of wavelengths that correspond to the uncorrected spectral data.
- **stationary_line_core** (*float, optional, default=None*) – Wavelength of the stationary line core.
- **constant_wavelengths** (*array_like, ndim=1, optional, default= see description*) – The desired set of wavelengths that the spectral data should be rescaled to represent. It is assumed that these have constant spacing, but that may not be a requirement if you specify your own array. The default value is an array from the minimum to the maximum wavelength of *original_wavelengths* in constant steps of *delta_lambda*, overshooting the upper bound if the maximum wavelength has not been reached.
- **delta_lambda** (*float, optional, default=0.05*) – The step used between each value of *constant_wavelengths* when its default value has to be calculated.
- **sigma** (*optional, default=None*) – Sigma values used to weight the fit. This attribute should be set by a child class of *ModelBase*.
- **prefilter_response** (*array_like, length=n_wavelengths, optional, default= see note*) – Each constant wavelength scaled spectrum will be corrected by dividing it by this array. If *prefilter_response* is not given, and *prefilter_ref_main* and *prefilter_ref_wvscl* are not given, *prefilter_response* will have a default value of *None*.
- **prefilter_ref_main** (*array_like, optional, default= None*) – If *prefilter_response* is not specified, this will be used along with *prefilter_ref_wvscl* to generate the default value of *prefilter_response*.
- **prefilter_ref_wvscl** (*array_like, optional, default=None*) – If *prefilter_response* is not specified, this will be used along with *prefilter_ref_main* to generate the default value of *prefilter_response*.
- **config** (*str, optional, default=None*) – Filename of a *.yml* file (relative to current directory) containing the initialising parameters for this object. Parameters provided explicitly to the object upon initialisation will override any provided in this file. All (or some) parameters that this object accepts can be specified in this file, except *neural_network* and *config*. Each line of the file should specify a different parameter and be formatted like *emission_guess: '[-inf, wl-0.15, 1e-6, 1e-6]'* or *original_wavelengths: 'original.fits'* for example. When specifying a string, use 'inf' to represent *np.inf* and 'wl' to represent *stationary_line_core* as shown. If the string matches a file, *mcalf.utils.misc.load_parameter()* is used to load the contents of the file.
- **output** (*str, optional, default=None*) – If the program wants to output data, it will place it relative to the location specified by this parameter. Some methods will only save data to a file if this parameter is not *None*. Such cases will be documented where relevant.

original_wavelengths

One-dimensional array of wavelengths that correspond to the uncorrected spectral data.

Type
array_like

stationary_line_core

Wavelength of the stationary line core.

Type
float, optional, default=None

`neural_network`

The neural network classifier object that is used to classify spectra. This attribute should be set by a child class of *ModelBase*.

Type
optional, default=None

`constant_wavelengths`

The desired set of wavelengths that the spectral data should be rescaled to represent. It is assumed that these have constant spacing, but that may not be a requirement if you specify your own array. The default value is an array from the minimum to the maximum wavelength of *original_wavelengths* in constant steps of *delta_lambda*, overshooting the upper bound if the maximum wavelength has not been reached.

Type
array_like, ndim=1, optional, default= see description

`sigma`

Sigma values used to weight the fit. This attribute should be set by a child class of *ModelBase*.

Type
optional, default=None

`prefilter_response`

Each constant wavelength scaled spectrum will be corrected by dividing it by this array. If *prefilter_response* is not given, and *prefilter_ref_main* and *prefilter_ref_wvscl* are not given, *prefilter_response* will have a default value of *None*.

Type
array_like, length=n_wavelengths, optional, default= see note

`output`

If the program wants to output data, it will place it relative to the location specified by this parameter. Some methods will only save data to a file if this parameter is not *None*. Such cases will be documented where relevant.

Type
str, optional, default=None

`array`

Array holding spectra.

Type
numpy.ndarray, dimensions are ['time', 'row', 'column', 'spectra']

`background`

Array holding spectral backgrounds.

Type
numpy.ndarray, dimensions are ['time', 'row', 'column']

Attributes Summary

default_kwargs
default_modelbase_kwargs
stationary_line_core

Methods Summary

_curve_fit(model, spectrum, guess, sigma, ...)	scipy.optimize.curve_fit() wrapper with error handling.
_fit(spectrum[, classification, spectrum_index])	Fit a single spectrum for the given profile or classification.
_get_time_row_column([time, row, column])	Validate and infer the time, row and column index.
_load_data(array[, names, target])	Load a specified array into the model object.
_set_prefilter()	Set the *prefilter_response* parameter.
_validate_base_attributes()	Validate some of the object's attributes.
classify_spectra([time, row, column, ...])	Classify the specified spectra.
fit([time, row, column, spectrum, ...])	Fits the model to specified spectra.
fit_spectrum(spectrum, **kwargs)	Fits the specified spectrum array.
get_spectra([time, row, column, spectrum, ...])	Gets corrected spectra from the spectral array.
load_array(array[, names])	Load an array of spectra.
load_background(array[, names])	Load an array of spectral backgrounds.
test(X, y)	Test the accuracy of the trained neural network.
train(X, y)	Fit the neural network model to spectra matrix X and spectra labels y.

Attributes Documentation

default_kwargs = {'constant_wavelengths': None, 'delta_lambda': 0.05, 'original_wavelengths': None, 'output': None, 'prefilter_ref_main': None, 'prefilter_ref_wvscl': None, 'prefilter_response': None, 'sigma': None, 'stationary_line_core': None}

default_modelbase_kwargs = {'constant_wavelengths': None, 'delta_lambda': 0.05, 'original_wavelengths': None, 'output': None, 'prefilter_ref_main': None, 'prefilter_ref_wvscl': None, 'prefilter_response': None, 'sigma': None, 'stationary_line_core': None}

stationary_line_core

Methods Documentation

_curve_fit(*model*, *spectrum*, *guess*, *sigma*, *bounds*, *x_scale*, *time=None*, *row=None*, *column=None*, ***kwargs*)

`scipy.optimize.curve_fit()` wrapper with error handling.

Passes a certain set of parameters to the `scipy.optimize.curve_fit()` function and catches some typical errors, presenting a more specific warning message.

Parameters

- **model** (*callable*) – The model function, f(x, ...). It must take the *ModelBase.constant_wavelenghts* attribute as the first argument and the parameters to fit as separate remaining arguments.
- **spectrum** (*array_like*) – The dependent data, with length equal to that of the *ModelBase.constant_wavelengths* attribute.
- **guess** (*array_like, optional*) – Initial guess for the parameters to fit.
- **sigma** (*array_like*) – Determines the uncertainty in the *spectrum*. Used to weight certain regions of the spectrum.
- **bounds** (*2-tuple of array_like*) – Lower and upper bounds on each parameter.
- **x_scale** (*array_like*) – Characteristic scale of each parameter.
- **time** (*optional, default=None*) – The time index for error handling.
- **row** (*optional, default=None*) – The row index for error handling.
- **column** (*optional, default=None*) – The column index for error handling.

Returns

- **fitted_parameters** (*numpy.ndarray, length=n_parameters*) – The parameters that recreate the model fitted to the spectrum.
- **success** (*bool*) – Whether the fit was successful or an error had to be handled.

See also:

fit
General fitting method.

fit_spectrum
Explicit spectrum fitting method.

Notes

More details can be found in the documentation for `scipy.optimize.curve_fit()` and `scipy.optimize.least_squares()`.

_fit(*spectrum*, *classification=None*, *spectrum_index=None*, ***kwargs*)

Fit a single spectrum for the given profile or classification.

> **Warning:** This call signature and docstring specify how the *_fit* method must be implemented in each subclass of *ModelBase*. **It is not implemented in this class.**

Parameters

- **spectrum** (*numpy.ndarray, ndim=1, length=n_constant_wavelengths*) – The spectrum to be fitted.
- **classification** (*int, optional, default=None*) – Classification to determine the fitted profile to use.
- **spectrum_index** (*array_like or list or tuple, length=3, optional, default=None*) – The [time, row, column] index of the *spectrum* provided. Only used for error reporting.

Returns
result – Outcome of the fit returned in a *mcalf.models.FitResult* object.

Return type
mcalf.models.FitResult

See also:

fit
The recommended method for fitting spectra.

mcalf.models.FitResult
The object that the fit method returns.

Notes

This method is called for each requested spectrum by the `models.ModelBase.fit()` method. This is where most of the adjustments to the fitting method should be made. See other subclasses of *models.ModelBase* for examples of how to implement this method in a new subclass. See `models.ModelBase.fit()` for more information on how this method is called.

_get_time_row_column(*time=None*, *row=None*, *column=None*)

Validate and infer the time, row and column index.

Takes any time, row and column index given and if any are not specified, they are returned as 0 if the spectral array only has one value at its dimension. If there are multiple and no index is specified, an error is raised due to the ambiguity.

Parameters

- **time** (*optional, default=None*) – The time index.
- **row** (*optional, default=None*) – The row index.
- **column** (*optional, default=None*) – The column index.

Returns

- *time* – The corrected time index.
- *row* – The corrected row index.
- *column* – The corrected column index.

See also:

mcalf.utils.misc.make_iter
Make a variable iterable.

Notes

No type checking is done on the input indices so it can be anything but in most cases will need to be either an integer or iterable. The *mcalf.utils.misc.make_iter()* function can be used to make indices iterable.

_load_data(*array*, *names=None*, *target=None*)

Load a specified array into the model object.

Load *array* with dimension names *names* into the attribute specified by *target*.

Parameters

- **array** (*numpy.ndarray*) – The array to load.
- **names** (*list of str, length=`array.ndim`*) – List of dimension names for *array*. Valid dimension names depend on *target*.
- **target** (*{'array', 'background'}*) – The attribute to load the *array* into.

See also:

load_array
Load and array of spectra.

load_background
Load an array of spectral backgrounds.

_set_prefilter()

Set the *prefilter_response* parameter.

Deprecated since version 0.2: Prefilter response correction code, and *prefilter_response*, *prefilter_ref_main* and *prefilter_ref_wvscl*, may be removed in a later release of MCALF. Spectra should be fully processed before loading into MCALF.

This method should be called in a child class once *stationary_line_core* has been set.

_validate_base_attributes()

Validate some of the object's attributes.

Raises
ValueError – To signal that an attribute is not valid.

classify_spectra(*time=None*, *row=None*, *column=None*, *spectra=None*, *only_normalise=False*)

Classify the specified spectra.

Will also normalise each spectrum such that its intensity will range from zero to one.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, a numpy.ndarray, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectra** (*numpy.ndarray, optional, default=None*) – The explicit spectra to classify. If *only_normalise* is False, this must be 1D. However, if *only_normalise* is set to true, *spectra* can be of any dimension. It is assumed that the final dimension is wavelengths, so return shape will be the same as *spectra*, except with no final wavelengths dimension.

- **only_normalise** (*bool, optional, default=False*) – Whether the single spectrum given in *spectra* should not be interpolated and corrected. If set to true, the only processing applied to *spectra* will be a normalisation to be in range 0 to 1.

Returns

classifications – Array of classifications with the same time, row and column indices as *spectra*.

Return type

numpy.ndarray

See also:

train
Train the neural network.

test
Test the accuracy of the neural network.

get_spectra
Get processed spectra from the objects *array* attribute.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8542.1, 8542.2, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load a trained neural network:

```
>>> import pickle
>>> pkl = open('trained_neural_network.pkl', 'rb')
>>> model.neural_network = pickle.load(pkl)
```

Classify an individual spectrum:

```
>>> spectrum = np.random.rand(30)
>>> model.classify_spectra(spectra=spectrum)
array([2])
```

When `only_normalise=True`, classify an n-dimensional spectral array:

```
>>> spectra = np.random.rand(5, 4, 3, 2, 30)
>>> model.classify_spectra(spectra=spectra, only_normalise=True).shape
(5, 4, 3, 2)
```

Load spectra from a file and classify:

```
>>> from astropy.io import fits
>>> spectra = fits.open('spectra_0000.fits')[0].data
>>> model.load_array(spectra, names=['wavelength', 'column', 'row'])
>>> model.classify_spectra(column=range(10, 15), row=[7, 16])
```

(continues on next page)

(continued from previous page)

```
array([[[0, 2, 0, 3, 0],
        [4, 0, 1, 0, 0]]])
```

fit(*time=None*, *row=None*, *column=None*, *spectrum=None*, *classifications=None*, *background=None*, *n_pools=None*, ***kwargs*)

Fits the model to specified spectra.

Fits the model to an array of spectra using multiprocessing if requested.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, `numpy.ndarray`, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectrum** (*numpy.ndarray, ndim=1, optional, default=None*) – The explicit spectrum to fit the model to.
- **classifications** (*int or array_like, optional, default=None*) – Classifications to determine the fitted profile to use. Will use neural network to classify them if not. If a multidimensional array, must have the same shape as [*time*, *row*, *column*]. Dimensions that would have length of 1 can be excluded.
- **background** (*float, optional, default=None*) – If provided, this value will be subtracted from the explicit spectrum provided in *spectrum*. Will not be applied to spectra found from the indices, use the *load_background()* method instead.
- **n_pools** (*int, optional, default=None*) – The number of processing pools to calculate the fitting over. This allocates the fitting of different spectra to *n_pools* separate worker processes. When processing a large number of spectra this will make the fitting process take less time overall. It also distributes such that each worker process has the same ratio of classifications to process. This should balance out the workload between workers. If few spectra are being fitted, performance may decrease due to the overhead associated with splitting the evaluation over separate processes. If *n_pools* is not an integer greater than zero, it will fit the spectrum with a for loop.
- ****kwargs** – Extra keyword arguments to pass to *_fit()*.

Returns

result – Outcome of the fits returned as a list of *FitResult* objects.

Return type

list of *FitResult*, length=n_spectra

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Set up the neural network classifier:

```
>>> model.neural_network = ...  # load an untrained classifier
>>> model.train(...)
>>> model.test(...)
```

Load the spectra and background array:

```
>>> model.load_array(...)
>>> model.load_background(...)
```

Fit a subset of the loaded spectra, using 5 processing pools:

```
>>> fits = model.fit(row=range(3, 5), column=range(200), n_pools=5)
>>> fits
['Successful FitResult with ________ profile of classification 0',
 'Successful FitResult with ________ profile of classification 2',
 ...
 'Successful FitResult with ________ profile of classification 0',
 'Successful FitResult with ________ profile of classification 4']
```

Merge the fit results into a *FitResults* object:

```
>>> results = mcalf.models.FitResults((500, 500), 8)
>>> for fit in fits:
...     results.append(fit)
```

See *fit_spectrum()* examples for how to manually providing a *spectrum* to fit.

fit_spectrum(*spectrum*, ***kwargs*)

Fits the specified spectrum array.

Passes the spectrum argument to the *fit()* method. For easily iterating over a list of spectra.

Parameters

- **spectrum** (*numpy.ndarray, ndim=1*) – The explicit spectrum.
- ****kwargs** – Extra keyword arguments to pass to *fit()*.

Returns

result – Result of the fit.

Return type

FitResult

See also:

fit

General fitting method.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Quickly provide a spectrum and fit it. Remember that the model must be optimised for the spectra that it is asked to fit. In this example the neural network is not called upon to classify the provided spectrum as a classification is provided directly:

```
>>> spectrum = np.random.rand(30)
>>> model.fit_spectrum(spectrum, classifications=0, background=142.2)
Successful FitResult with ________ profile of classification 0
```

As the spectrum is provided manually, any background value must also be provided manually. Alternatively, the background can be subtracted before passing to the function, as by default, no background is subtracted:

```
>>> model.fit_spectrum(spectrum - 142.2, classifications=0)
Successful FitResult with ________ profile of classification 0
```

get_spectra(*time=None*, *row=None*, *column=None*, *spectrum=None*, *correct=True*, *background=False*)

Gets corrected spectra from the spectral array.

Takes either a set of indices or an explicit spectrum and optionally applied corrections and background removal.

Parameters

- **time** (*int or iterable, optional, default=None*) – The time index. The index can be either a single integer index or an iterable. E.g. a list, a `numpy.ndarray`, a Python range, etc. can be used.
- **row** (*int or iterable, optional, default=None*) – The row index. See comment for *time* parameter.
- **column** (*int or iterable, optional, default=None*) – The column index. See comment for *time* parameter.
- **spectrum** (*ndarray of ndim=1, optional, default=None*) – The explicit spectrum. If provided, *time*, *row*, and *column* are ignored.
- **correct** (*bool, optional, default=True*) – Whether to reinterpolate the spectrum and apply the prefilter correction (if exists).
- **background** (*bool, optional, default=False*) – Whether to include the background in the outputted spectra. Only removes the background if the relevant background array has been loaded. Does not remove background is processing an explicit spectrum.

Returns

spectra

Return type

ndarray

Examples

Create a basic model:

```
>>> import mcalf.models
>>> import numpy as np
>>> wavelengths = np.linspace(8541.3, 8542.7, 30)
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Provide a single spectrum for processing, and notice output is 1D:

```
>>> spectrum = model.get_spectra(spectrum=np.random.rand(30))
>>> spectrum.ndim
1
```

Load an array of spectra:

```
>>> spectra = np.random.rand(3, 4, 30)
>>> model.load_array(spectra, names=['column', 'row', 'wavelength'])
```

Extract a single (unprocessed) spectrum from the loaded array, and notice output is 4D:

```
>>> spectrum = model.get_spectra(row=1, column=0, correct=False)
>>> spectrum.shape
(1, 1, 1, 30)
>>> (spectrum[0, 0, 0] == spectra[0, 1]).all()
True
```

Extract an array of spectra, and notice output is 4D, and with dimensions time, row, column, wavelength regardless of the original dimensions and order:

```
>>> spectrum = model.get_spectra(row=range(4), column=range(3))
>>> spectrum.shape
(1, 4, 3, 30)
```

Notice that the time index can be excluded, as the loaded array only represents a single time. However, in this case leaving out *row* or *column* results in an error as it is ambiguous:

```
>>> spectrum = model.get_spectra(row=range(4))
Traceback (most recent call last):
 ...
ValueError: column index must be specified as multiple indices exist
```

load_array(*array*, *names=None*)

Load an array of spectra.

Load *array* with dimension names *names* into the *array* parameter of the model object.

Parameters

- **array** (*numpy.ndarray, ndim>1*) – An array containing at least two spectra.
- **names** (*list of str, length=`array.ndim`*) – List of dimension names for *array*. Valid dimension names are 'time', 'row', 'column' and 'wavelength'. 'wavelength' is a required dimension.

See also:

load_background
Load an array of spectral backgrounds.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> from astropy.io import fits
>>> wavelengths = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0]
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load spectra from a file:

```
>>> spectra = fits.open('spectra_0000.fits')[0].data
>>> model.load_array(spectra, names=['wavelength', 'column', 'row'])
```

load_background(*array*, *names=None*)

Load an array of spectral backgrounds.

Load *array* with dimension names *names* into *background* parameter of the model object.

Parameters

- **array** (*numpy.ndarray, ndim>0*) – An array containing at least two backgrounds.
- **names** (*list of str, length=`array.ndim`*) – List of dimension names for *array*. Valid dimension names are 'time', 'row' and 'column'.

See also:

load_array
Load and array of spectra.

Examples

Create a basic model:

```
>>> import mcalf.models
>>> from astropy.io import fits
>>> wavelengths = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0]
>>> model = mcalf.models.ModelBase(original_wavelengths=wavelengths)
```

Load background array from a file:

```
>>> background = fits.open('background_0000.fits')[0].data
>>> model.load_background(background, names=['column', 'row'])
```

test(*X*, *y*)

Test the accuracy of the trained neural network.

Prints a table of results showing:

1) the percentage of predictions that equal the target labels;
2) the average classification deviation and standard deviation from the ground truth classification for each labelled classification;

3) the average classification deviation and standard deviation overall.

If the model object has an output parameter, it will create a CSV file (`output`/neural_network/test.csv) listing the predictions and ground truth data.

Parameters

- **X** (*numpy.ndarray or sparse matrix, shape=(n_spectra, n_wavelengths)*) – The input spectra.
- **y** (*numpy.ndarray, shape= (n_spectra,) or (n_spectra, n_outputs)*) – The target class labels.

See also:

train
Train the neural network.

train(*X*, *y*)

Fit the neural network model to spectra matrix X and spectra labels y.

Calls the *fit()* method on the *neural_network* parameter of the model object.

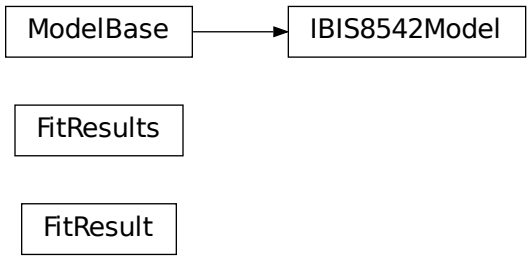
Parameters

- **X** (*numpy.ndarray or sparse matrix, shape=(n_spectra, n_wavelengths)*) – The input spectra.
- **y** (*numpy.ndarray, shape= (n_spectra,) or (n_spectra, n_outputs)*) – The target class labels.

See also:

test
Test how well the neural network has been trained.

Class Inheritance Diagram



1.2.3 MCALF profiles

This sub-package contains:

- Functions that can be used to model the spectra.
- Voigt profile with a variety of wrappers for different applications (*mcalf.profiles.voigt*).
- Gaussian profiles and skew normal distributions (*mcalf.profiles.gaussian*).

mcalf.profiles Package

mcalf.profiles.voigt Module

Functions

voigt_integrate(x, s, g[, clib])	Voigt function implementation (calculated by integrating).
voigt_faddeeva(x, s, g, **kwargs)	Voigt function implementation (Faddeeva).
voigt_mclean(x, s, g, **kwargs)	Voigt function implementation (efficient approximation).
voigt_nobg(x, a, b, s, g[, impl])	Voigt function with no background (Base Voigt function).
voigt(x, a, b, s, g, d, **kwargs)	Voigt function with background.
double_voigt_nobg(x, a1, b1, s1, g1, a2, b2, ...)	Double Voigt function with no background.
double_voigt(x, a1, b1, s1, g1, a2, b2, s2, ...)	Double Voigt function with background.

voigt_integrate

mcalf.profiles.voigt.**voigt_integrate**(*x*, *s*, *g*, *clib=False*, ***kwargs*)

Voigt function implementation (calculated by integrating).

The default Voigt implementation.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **s** (*float*) – Sigma (for Gaussian).
- **g** (*float*) – Gamma (for Lorentzian).
- **clib** (*bool, optional, default=True*) – Whether to use the complied C library or a slower Python version. If using the C library, the accuracy of the integration is reduced to give the code a significant speed boost. Python version can be used when speed is not a priority. Python version will remove deviations that are sometimes present around the wings due to the reduced accuracy. If the C extensions is not installed, will default to false.

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

Notes

More information on the Voigt function can be found here: https://en.wikipedia.org/wiki/Voigt_profile

voigt_faddeeva

mcalf.profiles.voigt.**voigt_faddeeva**(*x*, *s*, *g*, ***kwargs*)

Voigt function implementation (Faddeeva).

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **s** (*float*) – Sigma (for Gaussian).
- **g** (*float*) – Gamma (for Lorentzian).

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

voigt_mclean

mcalf.profiles.voigt.**voigt_mclean**(*x*, *s*, *g*, ***kwargs*)

Voigt function implementation (efficient approximation).

Not implemented in any models yet as initial tests exhibited slow convergence.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **s** (*float*) – Sigma (for Gaussian).
- **g** (*float*) – Gamma (for Lorentzian).

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

Notes

This algorithm is taken from A. B. McLean et al.[1].

[1] A. B. McLean, C. E. J. Mitchell and D. M. Swanston, "Implementation of an efficient analytical approximation to the Voigt function for photoemission lineshape analysis," Journal of Electron Spectroscopy and Related Phenomena, vol. 69, pp. 125-132, 1994. https://doi.org/10.1016/0368-2048(94)02189-7

References

voigt_nobg

mcalf.profiles.voigt.**voigt_nobg**(*x*, *a*, *b*, *s*, *g*, *impl=<function voigt_faddeeva>*, ***kwargs*)

Voigt function with no background (Base Voigt function).

This is the base of all the other Voigt functions.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **a** (*float*) – Amplitude of the Lorentzian.
- **b** (*float*) – Central line core.
- **s** (*float*) – Sigma (for Gaussian).
- **g** (*float*) – Gamma (for Lorentzian).

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

See also:

voigt
Voigt function with background added.

double_voigt_nobg
Two Voigt functions added together.

double_voigt
Two Voigt function and a background added together.

voigt

mcalf.profiles.voigt.**voigt**(*x*, *a*, *b*, *s*, *g*, *d*, ***kwargs*)

Voigt function with background.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **a** (*float*) – Amplitude of the Lorentzian.
- **b** (*float*) – Central line core.
- **s** (*float*) – Sigma (for Gaussian).
- **g** (*float*) – Gamma (for Lorentzian).
- **d** (*float*) – Background.

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

See also:

voigt_nobg
Base Voigt function with no background.

double_voigt_nobg
Two Voigt functions added together.

double_voigt
Two Voigt function and a background added together.

double_voigt_nobg

mcalf.profiles.voigt.**double_voigt_nobg**(*x*, *a1*, *b1*, *s1*, *g1*, *a2*, *b2*, *s2*, *g2*, ***kwargs*)

Double Voigt function with no background.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **a1** (*float*) – Amplitude of 1st Voigt function.
- **b1** (*float*) – Central line core of 1st Voigt function.
- **s1** (*float*) – Sigma (for Gaussian) of 1st Voigt function.
- **g1** (*float*) – Gamma (for Lorentzian) of 1st Voigt function.
- **a2** (*float*) – Amplitude of 2nd Voigt function.
- **b2** (*float*) – Central line core of 2nd Voigt function.
- **s2** (*float*) – Sigma (for Gaussian) of 2nd Voigt function.
- **g2** (*float*) – Gamma (for Lorentzian) of 2nd Voigt function.

Returns
result – The value of the Voigt function here.

Return type
numpy.ndarray, shape=`x.shape`

See also:

voigt_nobg
Base Voigt function with no background.

voigt
Voigt function with background added.

double_voigt
Two Voigt function and a background added together.

double_voigt

mcalf.profiles.voigt.**double_voigt**(*x*, *a1*, *b1*, *s1*, *g1*, *a2*, *b2*, *s2*, *g2*, *d*, ***kwargs*)

Double Voigt function with background.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Voigt function at.
- **a1** (*float*) – Amplitude of 1st Voigt function.
- **b1** (*float*) – Central line core of 1st Voigt function.
- **s1** (*float*) – Sigma (for Gaussian) of 1st Voigt function.
- **g1** (*float*) – Gamma (for Lorentzian) of 1st Voigt function.
- **a2** (*float*) – Amplitude of 2nd Voigt function.
- **b2** (*float*) – Central line core of 2nd Voigt function.
- **s2** (*float*) – Sigma (for Gaussian) of 2nd Voigt function.
- **g2** (*float*) – Gamma (for Lorentzian) of 2nd Voigt function.
- **d** (*float*) – Background.

Returns

result – The value of the Voigt function here.

Return type

numpy.ndarray, shape=`x.shape`

See also:

voigt_nobg
Base Voigt function with no background.

voigt
Voigt function with background added.

double_voigt_nobg
Two Voigt functions added together.

mcalf.profiles.gaussian Module

Functions

single_gaussian(x, a, b, c, d)	Gaussian function.

single_gaussian

mcalf.profiles.gaussian.**single_gaussian**(*x*, *a*, *b*, *c*, *d*)

Gaussian function.

Parameters

- **x** (*numpy.ndarray*) – Wavelengths to evaluate Gaussian function at.
- **a** (*float*) – Amplitude.
- **b** (*float*) – Central line core.
- **c** (*float*) – Sigma of Gaussian.
- **d** (*float*) – Background to add.

Returns

result – The value of the Gaussian function here.

Return type

numpy.ndarray, shape=`x.shape`

1.2.4 MCALF visualisation

This sub-package contains:

- Functions to plot the input spectrum and the fitted model.
- Functions to plot the spatial distribution and their general profile.
- Functions to plot the velocities calculated for a spectral imaging scan.

mcalf.visualisation Package

Functions

bar([class_map, vmin, vmax, reduce, style, ...])	Plot a bar chart of the classification abundances.
init_class_data(class_map[, vmin, vmax, ...])	Initialise dictionary of common classification plotting data.
plot_class_map([class_map, vmin, vmax, ...])	Plot a map of the classifications.
plot_classifications(spectra, labels[, ...])	Plot spectra grouped by their labelled classification.
plot_ibis8542(wavelengths, spectrum[, fit, ...])	Plot an *IBIS8542Model* fit.
plot_map(arr[, mask, umbra_mask, ...])	Plot a velocity map array.
plot_spectrum(wavelengths, spectrum[, ...])	Plot a spectrum with the wavelength grid shown.

bar

mcalf.visualisation.**bar**(*class_map=None*, *vmin=None*, *vmax=None*, *reduce=True*, *style='original'*, *cmap=None*, *ax=None*, *data=None*)

Plot a bar chart of the classification abundances.

Parameters

- **class_map** (*numpy.ndarray[int], ndim=2 or 3*) – Array of classifications. If the array is three-dimensional, it is assumed that the first dimension is time, and a time average classification will be plotted. The time average is the most common positive (valid) classification at each pixel.
- **vmin** (*int, optional, default=None*) – Minimum classification integer to plot. Must be greater or equal to zero. Defaults to min positive integer in *class_map*.
- **vmax** (*int, optional, default=None*) – Maximum classification integer to plot. Must be greater than zero. Defaults to max positive integer in *class_map*.
- **reduce** (*bool, optional, default=True*) – Whether to perform the time average described in *class_map* info.
- **style** (*str, optional, default='original'*) – The named matplotlib colormap to extract a `ListedColormap` from. Colours are selected from *vmin* to *vmax* at equidistant values in the range [0, 1]. The `ListedColormap` produced will also show bad classifications and classifications out of range in grey. The default 'original' is a special case used since early versions of this code. It is a hardcoded list of 5 colours. When the number of classifications exceeds 5, `style='viridis'` will be used.
- **cmap** (*str or matplotlib.colors.Colormap, optional, default=None*) – Parameter to pass to matplotlib.axes.Axes.imshow. This parameter overrides any cmap requested via the *style* parameter.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the velocity map will be plotted. Defaults to the current axis of the current figure.
- **data** (*dict, optional, default=None*) – Dictionary of common classification plotting settings generated by *init_class_data()*. If present, all other parameters are ignored except and *ax*.

Returns

b – The object returned by `matplotlib.axes.Axes.bar()` after plotting abundances.

Return type

matplotlib.container.BarContainer

See also:

mcalf.models.ModelBase.classify_spectra
Classify spectra.

mcalf.utils.smooth.average_classification
Average a 3D array of classifications.

Notes

Visualisation assumes that all integers between *vmin* and *vmax* are valid classifications, even if they do not appear in *class_map*.

Examples

- *Plot a bar chart of classifications*
- *Combine multiple classification plots*

init_class_data

mcalf.visualisation.**init_class_data**(*class_map*, *vmin=None*, *vmax=None*, *reduce=True*, *resolution=None*, *offset=(0, 0)*, *dimension='distance'*, *style='original'*, *cmap=None*, *colorbar_settings=None*, *ax=None*)

Initialise dictionary of common classification plotting data.

Parameters

- **class_map** (*numpy.ndarray[int], ndim=2 or 3*) – Array of classifications. If *reduce* is True (default) and the array is three-dimensional, it is assumed that the first dimension is time, and a time average classification will be calculated. The time average is the most common positive (valid) classification at each pixel.
- **vmin** (*int, optional, default=None*) – Minimum classification integer to include. Must be greater or equal to zero. Defaults to min positive integer in *class_map*. Classifications below this value will be set to -1.
- **vmax** (*int, optional, default=None*) – Maximum classification integer to include. Must be greater than zero. Defaults to max positive integer in *class_map*. Classifications above this value will be set to -1.
- **reduce** (*bool, optional, default=True*) – Whether to perform the time average described in *class_map* info.
- **resolution** (*tuple[float] or astropy.units.quantity.Quantity, optional, default=None*) – A 2-tuple (x, y) containing the length of each pixel in the x and y direction respectively. If a value has type `astropy.units.quantity.Quantity`, its axis label will include its attached unit, otherwise the unit will default to Mm. If *resolution* is None, both axes will be ticked with the default pixel value with no axis labels.
- **offset** (*tuple[float] or int, length=2, optional, default=(0, 0)*) – Two offset values (x, y) for the x and y axis respectively. Number of pixels from the 0 pixel to the first pixel. Defaults to the first pixel being at 0 length units. For example, in a 1000 pixel wide dataset, setting offset to -500 would place the 0 Mm location at the centre.
- **dimension** (*str or tuple[str] or list[str], length=2, optional, default='distance'*) – If an *ax* (and *resolution*) is provided, use this string as the *dimension name* that appears before the `(unit)` in the axis label. A 2-tuple (x, y) or list [x, y] can instead be given to provide a different name for the x-axis and y-axis respectively.
- **style** (*str, optional, default='original'*) – The named matplotlib colormap to extract a `ListedColormap` from. Colours are selected from *vmin* to *vmax* at equidistant values in the range [0, 1]. The `ListedColormap` produced will also show bad classifications and classifications out of range in grey. The default 'original' is a special case used since early

versions of this code. It is a hardcoded list of 5 colours. When the number of classifications exceeds 5, `style='viridis'` will be used.

- **cmap** (*str or matplotlib.colors.Colormap, optional, default=None*) – Parameter to pass to matplotlib.axes.Axes.imshow. This parameter overrides any cmap requested via the *style* parameter.
- **colorbar_settings** (*dict, optional, default=None*) – Dictionary of keyword arguments to pass to `matplotlib.figure.Figure.colorbar()`.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the classification map will be plotted. Defaults to the current axis of the current figure.

Returns
data – Common classification plotting settings.

Return type
dict

See also:

mcalf.visualisation.bar
Plot a bar chart of the classification abundances.

mcalf.visualisation.plot_class_map
Plot a map of the classifications.

mcalf.utils.smooth.mask_classifications
Mask 2D and 3D arrays of classifications.

mcalf.utils.plot.calculate_extent
Calculate the extent from a particular data shape and resolution.

mcalf.utils.plot.class_cmap
Create a listed colormap for a specific number of classifications.

Examples

- *Combine multiple classification plots*

plot_class_map

mcalf.visualisation.**plot_class_map**(*class_map=None*, *vmin=None*, *vmax=None*, *resolution=None*, *offset=(0, 0)*, *dimension='distance'*, *style='original'*, *cmap=None*, *show_colorbar=True*, *colorbar_settings=None*, *ax=None*, *data=None*)

Plot a map of the classifications.

Parameters

- **class_map** (*numpy.ndarray[int], ndim=2 or 3*) – Array of classifications. If the array is three-dimensional, it is assumed that the first dimension is time, and a time average classification will be plotted. The time average is the most common positive (valid) classification at each pixel.
- **vmin** (*int, optional, default=None*) – Minimum classification integer to plot. Must be greater or equal to zero. Defaults to min positive integer in *class_map*.
- **vmax** (*int, optional, default=None*) – Maximum classification integer to plot. Must be greater than zero. Defaults to max positive integer in *class_map*.

- **resolution** (*tuple[float] or astropy.units.quantity.Quantity, optional, default=None*) – A 2-tuple (x, y) containing the length of each pixel in the x and y direction respectively. If a value has type `astropy.units.quantity.Quantity`, its axis label will include its attached unit, otherwise the unit will default to Mm. If *resolution* is None, both axes will be ticked with the default pixel value with no axis labels.
- **offset** (*tuple[float] or int, length=2, optional, default=(0, 0)*) – Two offset values (x, y) for the x and y axis respectively. Number of pixels from the 0 pixel to the first pixel. Defaults to the first pixel being at 0 length units. For example, in a 1000 pixel wide dataset, setting offset to -500 would place the 0 Mm location at the centre.
- **dimension** (*str or tuple[str] or list[str], length=2, optional, default='distance'*) – If an *ax* (and *resolution*) is provided, use this string as the *dimension name* that appears before the `(unit)` in the axis label. A 2-tuple (x, y) or list [x, y] can instead be given to provide a different name for the x-axis and y-axis respectively.
- **style** (*str, optional, default='original'*) – The named matplotlib colormap to extract a `ListedColormap` from. Colours are selected from *vmin* to *vmax* at equidistant values in the range [0, 1]. The `ListedColormap` produced will also show bad classifications and classifications out of range in grey. The default 'original' is a special case used since early versions of this code. It is a hardcoded list of 5 colours. When the number of classifications exceeds 5, `style='viridis'` will be used.
- **cmap** (*str or matplotlib.colors.Colormap, optional, default=None*) – Parameter to pass to matplotlib.axes.Axes.imshow. This parameter overrides any cmap requested via the *style* parameter.
- **show_colorbar** (*bool, optional, default=True*) – Whether to draw a colorbar.
- **colorbar_settings** (*dict, optional, default=None*) – Dictionary of keyword arguments to pass to `matplotlib.figure.Figure.colorbar()`. Ignored if *show_colorbar* is False.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the velocity map will be plotted. Defaults to the current axis of the current figure.
- **data** (*dict, optional, default=None*) – Dictionary of common classification plotting settings generated by *init_class_data()*. If present, all other parameters are ignored except *show_colorbar* and *ax*.

Returns

im – The object returned by `matplotlib.axes.Axes.imshow()` after plotting *class_map*.

Return type

matplotlib.image.AxesImage

See also:

mcalf.models.ModelBase.classify_spectra
Classify spectra.

mcalf.utils.smooth.average_classification
Average a 3D array of classifications.

Notes

Visualisation assumes that all integers between *vmin* and *vmax* are valid classifications, even if they do not appear in *class_map*.

Examples

- *Working with IBIS data*
- *Using IBIS8542Model*
- *Combine multiple classification plots*
- *Plot a map of classifications*

plot_classifications

mcalf.visualisation.**plot_classifications**(*spectra*, *labels*, *nrows=None*, *ncols=None*, *nlines=20*, *style='original'*, *cmap=None*, *show_labels=True*, *plot_settings={}*, *fig=None*)

Plot spectra grouped by their labelled classification.

Parameters

- **spectra** (*ndarray, ndim=2*) – Two-dimensional array with dimensions [spectra, wavelengths].
- **labels** (ndarray, ndim=1, length of *spectra*) – List of classifications for each spectrum in *spectra*.
- **nrows** (*int, optional, default=None*) – Number of rows. Defaults to rows of max width 3 axes. Special case: four plots will be in a 2x2 grid. Only one of *nrows* and *ncols* can be specified.
- **ncols** (*int, optional, default=None*) – Number of columns. Defaults to rows of max width 3 axes. Special case: four plots will be in a 2x2 grid. Only one of *nrows* and *ncols* can be specified.
- **nlines** (*int, optional, default=20*) – Maximum number of lines per classification plot.
- **style** (*str, optional, default='original'*) – The named matplotlib colormap to extract a `ListedColormap` from. Colours are selected from *vmin* to *vmax* at equidistant values in the range [0, 1]. The `ListedColormap` produced will also show bad classifications and classifications out of range in grey. The default 'original' is a special case used since early versions of this code. It is a hardcoded list of 5 colours. When the number of classifications exceeds 5, `style='viridis'` will be used.
- **cmap** (*callable, optional, default=None*) – Function that returns a colour for each input from zero to num. classifications. This parameter overrides any cmap requested via the *style* parameter. Return value is passed to the *color* parameter of `matplotlib.pyplot.axes.Axes.plot()`.
- **show_labels** (*bool, optional, default=True*) – Whether to label the axes with the corresponding classifications.

- **plot_settings** (*dict, optional, default={}*) – Dictionary of keyword arguments to pass to `matplotlib.pyplot.axes.Axes.plot()`.
- **fig** (*matplotlib.figure.Figure, optional, default=None*) – Figure into which the classifications will be plotted. Defaults to the current figure.

Returns
gs – The grid layout subplots are placed on within the figure.

Return type
matplotlib.gridspec.GridSpec

Examples

- *Working with IBIS data*
- *Using IBIS8542Model*
- *Combine multiple classification plots*
- *Plot a grid of spectra grouped by classification*

plot_ibis8542

mcalf.visualisation.**plot_ibis8542**(*wavelengths*, *spectrum*, *fit=None*, *background=0*, *sigma=None*, *sigma_scale=70*, *stationary_line_core=None*, *impl=<function voigt_faddeeva>*, *subtraction=False*, *separate=False*, *show_intensity=True*, *show_legend=True*, *ax=None*)

Plot an *IBIS8542Model* fit.

Note: It is recommended to use the plot method built into either the *IBIS8542Model* class or the *FitResult* class instead.

Parameters

- **wavelengths** (*numpy.ndarray*) – The x-axis values.
- **spectrum** (*numpy.ndarray, length=n_wavelengths*) – The y-axis values.
- **fit** (*array_like, optional, default=None*) – The fitted parameters.
- **background** (*float or numpy.ndarray, length=n_wavelengths, optional, default=0*) – The background to add to the fitted profiles.
- **sigma** (*numpy.ndarray, length=n_wavelengths, optional, default=None*) – The sigma profile used when fitting the parameters to *spectrum*. If given, will be plotted as shaded regions.
- **sigma_scale** (*float, optional, default=70*) – A factor to multiply the error bars to change their prominence.
- **stationary_line_core** (*float, optional, default=None*) – If given, will show a dashed line at this wavelength.
- **impl** (*callable, optional, default=voigt_faddeeva*) – The Voigt implementation to use.

- **subtraction** (*bool, optional, default=False*) – Whether to plot the *spectrum* minus emission fit (if exists) instead.
- **separate** (*bool, optional, default=False*) – Whether to plot the fitted profiles separately (if multiple components exist).
- **show_intensity** (*bool, optional, default=True*) – Whether to show the intensity axis tick labels and axis label.
- **show_legend** (*bool, optional, default=True*) – Whether to draw a legend on the axes.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the fit will be plotted. Defaults to the current axis of the current figure.

Returns
ax – Axes the lines are drawn on.

Return type
matplotlib.axes.Axes

See also:

mcalf.models.IBIS8542Model.plot
General plotting method.

mcalf.models.IBIS8542Model.plot_separate
Plot the fit parameters separately.

mcalf.models.IBIS8542Model.plot_subtraction
Plot the spectrum with the emission fit subtracted from it.

mcalf.models.FitResult.plot
Plotting method provided by the fit result.

Examples

- *Plot a fitted spectrum*

plot_map

mcalf.visualisation.**plot_map**(*arr*, *mask=None*, *umbra_mask=None*, *resolution=None*, *offset=(0, 0)*, *vmin=None*, *vmax=None*, *lw=None*, *show_colorbar=True*, *unit='km/s'*, *ax=None*)

Plot a velocity map array.

Parameters

- **arr** (*numpy.ndarray[float] or astropy.units.quantity.Quantity, ndim=2*) – Two-dimensional array of velocities.
- **mask** (*numpy.ndarray[bool], ndim=2, shape=arr, optional, default=None*) – Mask showing the region where velocities were found for. True is outside the velocity region and False is where valid velocities should be found. Specifying a mask allows for errors in the velocity calculation to be black and points outside the region to be gray. If omitted, all invalid points will be gray.

- **umbra_mask** (*numpy.ndarray[bool], ndim=2, shape=arr, optional, default=None*) – A mask of the umbra, True outside, False inside. If given, a contour will outline the umbra, or other feature the mask represents.
- **resolution** (*tuple[float] or astropy.units.quantity.Quantity, optional, default=None*) – A 2-tuple (x, y) containing the length of each pixel in the x and y direction respectively. If a value has type `astropy.units.quantity.Quantity`, its axis label will include its attached unit, otherwise the unit will default to Mm. If *resolution* is None, both axes will be ticked with the default pixel value with no axis labels.
- **offset** (*tuple[float] or int, length=2, optional, default=(0, 0)*) – Two offset values (x, y) for the x and y axis respectively. Number of pixels from the 0 pixel to the first pixel. Defaults to the first pixel being at 0 length units. For example, in a 1000 pixel wide dataset, setting offset to -500 would place the 0 Mm location at the centre.
- **vmin** (float, optional, default= `-max(|arr|)`) – Minimum velocity to plot. If not given, will be -vmax, for vmax not None.
- **vmax** (float, optional, default= `max(|arr|)`) – Maximum velocity to plot. If not given, will be -vmin, for vmin not None.
- **lw** (*float, optional, default=None*) – The line width of the contour line plotted for *umbra_mask*. Passed as *linewidths* to `matplotlib.axes.Axes.contour()`.
- **show_colorbar** (*bool, optional, default=True*) – Whether to draw a colorbar.
- **unit** (*str or astropy.units.UnitBase or astropy.units.quantity.Quantity, optional, default='km/s'*) – The units of *arr* data. Printed on colorbar.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the velocity map will be plotted. Defaults to the current axis of the current figure.

Returns

im – The object returned by `matplotlib.axes.Axes.imshow()` after plotting *arr*.

Return type

matplotlib.image.AxesImage

See also:

mcalf.models.FitResults.velocities

Calculate the Doppler velocities for an array of fits.

Examples

- *Working with IBIS data*
- *Using IBIS8542Model*
- *Plot a map of velocities*

plot_spectrum

mcalf.visualisation.**plot_spectrum**(*wavelengths*, *spectrum*, *normalised=True*, *smooth=True*, *ax=None*)

Plot a spectrum with the wavelength grid shown.

Intended for plotting the raw data.

Parameters

- **wavelengths** (*numpy.ndarray*) – The x-axis values.
- **spectrum** (*numpy.ndarray, length=n_wavelengths*) – The y-axis values.
- **normalised** (*bool, optional, default=True*) – Whether to normalise the spectrum using the last three spectral points.
- **smooth** (*bool, optional, default=True*) – Whether to smooth the *spectrum* with a spline.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which the fit will be plotted. Defaults to the current axis of the current figure.

Returns

ax – Axes the lines are drawn on.

Return type

matplotlib.axes.Axes

Examples

- *Using IBIS8542Model*
- *Plot a fitted spectrum*
- *Plot a spectrum*

1.2.5 MCALF utils

This sub-package contains:

- Functions for processing spectra (*mcalf.utils.spec*).
- Functions for smoothing n-dimensional arrays (*mcalf.utils.smooth*).
- Functions for masking the input data to limit the region computed (*mcalf.utils.mask*).
- Functions for helping with plotting (*mcalf.utils.plot*).
- Classes for managing collections of data (*mcalf.utils.collections*).
- Miscellaneous utility functions (*mcalf.utils.misc*).

mcalf.utils Package

mcalf.utils.spec Module

Functions

reinterpolate_spectrum(spectrum, ...)	Reinterpolate the spectrum.
normalise_spectrum(spectrum[, ...])	Normalise an individual spectrum to have intensities in range [0, 1].
generate_sigma(sigma_type, wavelengths, ...)	Generate the default sigma profiles.

reinterpolate_spectrum

mcalf.utils.spec.**reinterpolate_spectrum**(*spectrum*, *original_wavelengths*, *constant_wavelengths*)

Reinterpolate the spectrum.

Reinterpolates the spectrum such that intensities at *original_wavelengths* are transformed into intensities at *constant_wavelengths*. Uses `scipy.interpolate.InterpolatedUnivariateSpline` to interpolate.

Parameters

- **spectrum** (*numpy.ndarray, ndim=1*) – Spectrum to reinterpolate.
- **original_wavelengths** (numpy.ndarray, ndim=1, length=length of *spectrum*) – Wavelengths of *spectrum*.
- **constant_wavelengths** (*numpy.ndarray, ndim=1*) – Wavelengths to cast *spectrum* into.

Returns

spectrum – Reinterpolated spectrum.

Return type

numpy.ndarray, length=length of *constant_wavelengths*

normalise_spectrum

mcalf.utils.spec.**normalise_spectrum**(*spectrum*, *original_wavelengths=None*, *constant_wavelengths=None*, *prefilter_response=None*, *model=None*)

Normalise an individual spectrum to have intensities in range [0, 1].

Warning: Not recommended for normalising many spectra in a loop.

Parameters

- **spectrum** (*numpy.ndarray, ndim=1*) – Spectrum to reinterpolate and normalise.
- **original_wavelengths** (numpy.ndarray, ndim=1, length=length of *spectrum*, optional) – Wavelengths of *spectrum*.
- **constant_wavelengths** (*numpy.ndarray, ndim=1, optional*) – Wavelengths to cast *spectrum* into.

- **prefilter_response** (numpy.ndarray, ndim=1, length=length of *constant_wavelengths*, optional) – Prefilter response to divide spectrum by.
- **model** (*child class of* mcalf.models.ModelBase, *optional*) – Model to extract the above parameters from.

Returns
spectrum – The normalised spectrum.

Return type
numpy.ndarray, ndim-1, length=length of *constant_wavelengths*

generate_sigma

mcalf.utils.spec.**generate_sigma**(*sigma_type*, *wavelengths*, *line_core*, *a=-0.95*, *c=0.04*, *d=1*, *centre_rad=7*, *a_peak=0.4*)

Generate the default sigma profiles.

Parameters

- **sigma_type** (*int*) – Type of profile to generate. Should be either *1* or *2*.
- **wavelengths** (*array_like*) – Wavelengths to use for sigma profile.
- **line_core** (*float*) – Line core to use as centre of Gaussian sigma profile.
- **a** (*float, optional, default=-0.95*) – Amplitude of Gaussian sigma profile.
- **c** (*float, optional, default=0.04*) – Sigma of Gaussian sigma profile.
- **d** (*float, optional, default=1*) – Background of Gaussian sigma profile.
- **centre_rad** (*int, optional, default=7*) – Width of central flattened region.
- **a_peak** (*float, optional, default=0.4*) – Amplitude of central 7 pixel section, if *sigma_type* is 2.

Returns
sigma – The generated sigma profile.

Return type
numpy.ndarray, length=n_wavelengths

mcalf.utils.smooth Module

Functions

moving_average(array, width)	Boxcar moving average.
gaussian_kern_3d([width, sigma])	3D Gaussian kernel.
smooth_cube(cube, mask, **kwargs)	Apply Gaussian smoothing to velocities.
mask_classifications(class_map[, vmin, ...])	Mask 2D and 3D arrays of classifications.

moving_average

mcalf.utils.smooth.**moving_average**(*array*, *width*)

Boxcar moving average.

Calculate the moving average of an array with a boxcar of defined width. An odd width is recommended.

Parameters

- **array** (*numpy.ndarray, ndim=1*) – Array to find the moving average of.
- **width** (*int*) – Width of the boxcar. Odd integer recommended. Less than or equal to length of *array*.

Returns

averaged – Averaged array.

Return type

numpy.ndarray, shape=`array`

Notes

The moving average is calculated at each point of the *array* by finding the (unweighted) mean of the subarrays of length given by *width*. These subarrays are centred at the point in the *array* that the current average is currently being calculated for. If an odd *width* is chosen, the sub array will include the current point plus an equal number of points on either side. However, if an even *width* is chosen, the sub array will bias including the extra point to the left of the current index. If the subarray spans past the boundaries, the values beyond the boundary is ignored and the mean is calculated by dividing by the number of points that are inside the boundaries.

Examples

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> moving_average(x, 3)
array([1.5, 2. , 3. , 4. , 4.5])
>>> moving_average(x, 2)
array([1. , 1.5, 2.5, 3.5, 4.5])
```

gaussian_kern_3d

mcalf.utils.smooth.**gaussian_kern_3d**(*width=5*, *sigma=(1, 1, 1)*)

3D Gaussian kernel.

Create a Gaussian kernel of shape *width* `*` *width* `*` *width*.

Parameters

- **width** (*int, optional, default=5*) – Length of all three dimensions of the Gaussian kernel.
- **sigma** (*array_like, tuple, optional, default=(1, 1, 1)*) – Sigma values for the time, horizontal and vertical dimensions.

Returns

kernel – The generated kernel.

Return type
numpy.ndarray, shape=(*width*, *width*, *width*)

Examples

```
>>> gaussian_kern_3d(width=3, sigma=(2, 1, 1.5))
array([[[0.42860385, 0.53526143, 0.42860385],
        [0.48567179, 0.60653066, 0.48567179],
        [0.42860385, 0.53526143, 0.42860385]],

       [[0.70664828, 0.8824969 , 0.70664828],
        [0.8007374 , 1.        , 0.8007374 ],
        [0.70664828, 0.8824969 , 0.70664828]],

       [[0.42860385, 0.53526143, 0.42860385],
        [0.48567179, 0.60653066, 0.48567179],
        [0.42860385, 0.53526143, 0.42860385]]])
```

smooth_cube

mcalf.utils.smooth.**smooth_cube**(*cube*, *mask*, ***kwargs*)

Apply Gaussian smoothing to velocities.

Smooth the cube of velocities with a Gaussian kernel, applying weights at boundaries.

Parameters

- **cube** (*numpy.ndarray, ndim=3*) – Cube of velocities with dimensions [time, row, column].
- **mask** (*numpy.ndarray, ndim=2*) – The mask to apply to the [row, column] at every time. Points that are 0 or false will be removed.
- ****kwargs** – Keyword arguments to pass to *gaussian_kern_3d()*.

Returns
cube_ – The smoothed cube.

Return type
numpy.ndarray, shape=`cube`

mask_classifications

mcalf.utils.smooth.**mask_classifications**(*class_map*, *vmin=None*, *vmax=None*, *reduce=True*)

Mask 2D and 3D arrays of classifications.

If 3D, also reduces to 2D by selecting the most common classification along the first dimension.

Parameters

- **class_map** (*numpy.ndarray[int], ndim=2 or 3*) – Array of classifications. If *reduce* is True (default) and the array is three-dimensional, it is assumed that the first dimension is time, and a time average classification will be calculated. The time average is the most common positive (valid) classification at each pixel.

- **vmin** (*int, optional, default=None*) – Minimum classification integer to include. Must be greater or equal to zero. Defaults to min positive integer in *class_map*. Classifications below this value will be set to -1.
- **vmax** (*int, optional, default=None*) – Maximum classification integer to include. Must be greater than zero. Defaults to max positive integer in *class_map*. Classifications above this value will be set to -1.
- **reduce** (*bool, optional, default=True*) – Whether to perform the time average described in *class_map* info.

Returns

- **class_map** (*numpy.ndarray[int], ndim=2*) – *class_map* with values between *vmin* and *vmax* averaged along the first dimension.
- **vmin** (*int*) – Updated *vmin* value.
- **vmax** (*int*) – Updated *vmax* value.

See also:

mcalf.visualisation.plot_class_map
Plot a map of the classifications.

mcalf.utils.mask Module

Functions

genmask(width, height[, radius, ...])	Generate a circular mask of specified size.
radial_distances(n_cols, n_rows)	Generates a 2D array of specified shape of radial distances from the centre.

genmask

mcalf.utils.mask.**genmask**(*width*, *height*, *radius=inf*, *right_shift=0*, *up_shift=0*)

Generate a circular mask of specified size.

Parameters

- **width** (*int*) – Width of mask.
- **height** (*int*) – Height of mask.
- **radius** (*int, optional, default=inf*) – Radius of mask.
- **right_shift** (*int, optional, default=0*) – Indices to shift forward through row.
- **up_shift** (*int, optional, default=0*) – Indices to shift forward through columns.

Returns
array – The generated mask.

Return type
numpy.ndarray, shape=(height, width)

Examples

- *Plot a map of velocities*

radial_distances

mcalf.utils.mask.**radial_distances**(*n_cols*, *n_rows*)

Generates a 2D array of specified shape of radial distances from the centre.

Parameters

- **n_cols** (*int*) – Number of columns.
- **n_rows** (*int*) – Number of rows.

Returns

array – Array of radial distances.

Return type

numpy.ndarray, shape=(n_rows, n_cols)

See also:

genmask

Generates a circular mask.

mcalf.utils.plot Module

Functions

hide_existing_labels(plot_settings[, axes, fig])	Hides labels for each dictionary provided if label already exists in legend.
calculate_axis_extent(resolution, px[, ...])	Calculate the extent from a resolution value along a particular axis.
calculate_extent(shape, resolution[, ...])	Calculate the extent from a particular data shape and resolution.
class_cmap(style, n)	Create a listed colormap for a specific number of classifications.

hide_existing_labels

mcalf.utils.plot.**hide_existing_labels**(*plot_settings*, *axes=None*, *fig=None*)

Hides labels for each dictionary provided if label already exists in legend.

Parameters

- **plot_settings** (*dict of {str: dict}*) – Dictionary of lines to be plotted. Values must be dictionaries with a 'label' entry that this function my append with a '_' to hide the label.
- **axes** (*list of matplotlib.axes.Axes, optional, default=None*) – List of axes to extract lines labels from. Extracts axes from *fig* if omitted.
- **fig** (*matplotlib.figure.Figure, optional, default=None*) – Figure to take line labels from. Uses current figure if omitted.

Notes

Only the `plot_settings[*]['label']` values are uses to assess if a label has already been used. Other *plot_settings* parameters such as *color* are ignored.

Examples

Import plotting package:

```
>>> import matplotlib.pyplot as plt
```

Define various plot settings:

```
>>> plot_settings = {
...     'LineA': {'color': 'r', 'label': 'A'},
...     'LineB': {'color': 'g', 'label': 'B'},
...     'LineC': {'color': 'b', 'label': 'C'},
... }
```

Create a figure and plot two lines on the first axes:

```
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot([0, 1], [0, 1], **plot_settings['LineA'])
[<matplotlib.lines.Line2D object at 0x...>]
>>> axes[0].plot([0, 1], [1, 0], **plot_settings['LineB'])
[<matplotlib.lines.Line2D object at 0x...>]
```

Set labels already used to be hidden if used again:

```
>>> hide_existing_labels(plot_settings)
```

Anything already used will have an underscore prepended:

```
>>> [x['label'] for x in plot_settings.values()]
['_A', '_B', 'C']
```

Plot two lines on the second axes:

```
>>> axes[1].plot([0, 1], [0, 1], **plot_settings['LineB'])  # Label hidden
[<matplotlib.lines.Line2D object at 0x...>]
>>> axes[1].plot([0, 1], [1, 0], **plot_settings['LineC'])
[<matplotlib.lines.Line2D object at 0x...>]
```

Show the figure with the legend:

```
>>> fig.legend(ncol=3, loc='upper center')
<matplotlib.legend.Legend object at 0x...>
>>> plt.show()
>>> plt.close()
```

calculate_axis_extent

mcalf.utils.plot.**calculate_axis_extent**(*resolution*, *px*, *offset=0*, *unit='Mm'*)

Calculate the extent from a resolution value along a particular axis.

Parameters

- **resolution** (*float or astropy.units.quantity.Quantity*) – Length of each pixel. Unit defaults to *unit* is not an astropy quantity.
- **px** (*int*) – Number of pixels extent is being calculated for.
- **offset** (*int or float, default=0*) – Number of pixels from the 0 pixel to the first pixel. Defaults to the first pixel being at 0 length units. For example, in a 1000 pixel wide dataset, setting offset to -500 would place the 0 Mm location at the centre.
- **unit** (*str, default="Mm"*) – Default unit string to use if *resolution* is not an astropy quantity.

Returns

- **first** (*float*) – First extent value.
- **last** (*float*) – Last extent value.
- **unit** (*str*) – Unit of extent values.

calculate_extent

mcalf.utils.plot.**calculate_extent**(*shape*, *resolution*, *offset=(0, 0)*, *ax=None*, *dimension=None*, ***kwargs*)

Calculate the extent from a particular data shape and resolution.

This function assumes a lower origin is being used with matplotlib.

Parameters

- **shape** (*tuple[int]*) – Shape (y, x) of the `numpy.ndarray` of the data being plotted. First integer corresponds to the y-axis and the second integer is for the x-axis.
- **resolution** (*tuple[float] or astropy.units.quantity.Quantity*) – A 2-tuple (x, y) containing the length of each pixel in the x and y direction respectively. If a value has type `astropy.units.quantity.Quantity`, its axis label will include its attached unit, otherwise the unit will default to Mm. The *ax* parameter must be specified to set its labels. If *resolution* is None, this function will immediately return None.
- **offset** (*tuple[float] or int, length=2, optional, default=(0, 0)*) – Two offset values (x, y) for the x and y axis respectively. Number of pixels from the 0 pixel to the first pixel. Defaults to the first pixel being at 0 length units. For example, in a 1000 pixel wide dataset, setting offset to -500 would place the 0 Mm location at the centre.
- **ax** (*matplotlib.axes.Axes, optional, default=None*) – Axes into which axis labels will be plotted. Defaults to not printing axis labels.
- **dimension** (*str or tuple[str] or list[str], length=2, optional, default=None*) – If an *ax* (and *resolution*) is provided, use this string as the *dimension name* that appears before the `(unit)` in the axis label. A 2-tuple (x, y) or list [x, y] can instead be given to provide a different name for the x-axis and y-axis respectively. Defaults is equivalent to `dimension=('x-axis', 'y-axis')`.
- ****kwargs** – Extra keyword arguments to pass to *calculate_axis_extent()*.

Returns
extent – The extent value that will be passed to matplotlib functions with a lower origin. Will return None if *resolution* is None.

Return type
tuple[float], length=4

class_cmap

mcalf.utils.plot.**class_cmap**(*style*, *n*)

Create a listed colormap for a specific number of classifications.

Parameters

- **style** (*str*) – The named matplotlib colormap to extract a `ListedColormap` from. Colours are selected from *vmin* to *vmax* at equidistant values in the range [0, 1]. The `ListedColormap` produced will also show bad classifications and classifications out of range in grey. The 'original' style is a special case used since early versions of this code. It is a hardcoded list of 5 colours. When the number of classifications exceeds 5, `style='viridis'` will be used.
- **n** (*int*) – Number of colours (i.e., number of classifications) to include in the colormap.

Returns
cmap – Colormap generated for classifications.

Return type
matplotlib.colors.ListedColormap

mcalf.utils.collections Module

Classes

Parameter(name[, value])	A named parameter with a optional value.
ParameterDict([dict])	An unordered dictionary of *Parameter* objects.
OrderedParameterDict	An ordered dictionary of *Parameter* objects.
BaseParameterDict()	A base class for dictionaries of *Parameter* objects.
SyncedParameters()	Database for keeping *Parameter* objects in sync.

Parameter

class mcalf.utils.collections.**Parameter**(*name*, *value=None*)

Bases: `object`

A named parameter with a optional value.

The value can be changed at any time and very basic operations can be queued and evaluated on demand.

Parameters

- **name** (*str*) – Name of parameter.
- **value** (*int or float, optional, default=None*) – Value of parameter. Other types may work but not supported.

name
Name of parameter.

Type
str

value
Value of parameter, without operations applied.

Type
int or float

operations
List of operations to apply.

Type
list

Raises

- **ValueError** – The parameter is evaluated without a value.
- **NotImplementedError** – Operations which require parentheses are applied.

See also:

Parameters
Collection of synced `Parameter` objects.

Notes

Basic operations can be applied to this object, i.e., `+`, `-`, `*` and `/`. The `Parameter` object must be the left-most entity in an expression. Expressions which require parentheses are not supported. The only types that are supported in expressions are `Parameter`, `float` and `int`. Any type should work as a value for `Parameter` (such as an array), however, only `float` and `int` are supported. See the examples for more details in how it works.

Examples

```
>>> Parameter('x')
'x'
>>> Parameter('x', 12)
'12'
>>> Parameter('x') + 1
'x+1'
>>> Parameter('x', 1) + 1
'x+1'
>>> a = Parameter('y') * 2
>>> a
'y*2'
>>> a.value = 5
>>> a == 10
True
>>> a.eval()
10
```

(continues on next page)

(continued from previous page)

```
>>> (a * 10).eval()
100
>>> a * 10 + 1.5
'y*2*10+1.5'
>>> (a + 10) * 2
Traceback (most recent call last):
 ...
NotImplementedError: Parameter equations with brackets are not supported.
```

Attributes Summary

value_or_name	Returns its value if set, otherwise returns its name.

Methods Summary

apply_operation(op, other)	Apply an operation to the `Parameter`.
copy()	Return a copy of the parameter.
eval()	Evaluate the expression using the parameter's current value.
verify()	Enforce order of operations.

Attributes Documentation

value_or_name

Returns its value if set, otherwise returns its name.

Methods Documentation

apply_operation(*op*, *other*)

Apply an operation to the `Parameter`.

Parameters

- **op** (*str*) – Operation symbol to apply. +, -, * or /.
- **other** (*float or int*) – Number on RHS.

Returns

A copy of `self` with the operation added to the queue.

Return type

obj

Notes

You should only use this method directly if a builtin Python operation is not otherwise implemented in this class, i.e., do `a + 1` and not `a.apply_operation('+', 1)`.

copy()

Return a copy of the parameter.

eval()

Evaluate the expression using the parameter's current value.

verify()

Enforce order of operations.

ParameterDict

class mcalf.utils.collections.**ParameterDict**(*dict=None*, */*, ***kwargs*)

Bases: *BaseParameterDict*, UserDict

An unordered dictionary of *Parameter* objects.

The same parameters existing across multiple dictionary values can be kept in sync with each other. For a *Parameter* object to be kept in sync it must be located in the dictionary such that `{key: Parameter}` or `{key: List[Parameter]}`.

Examples

```
>>> d = ParameterDict({
...     'a': Parameter('x') + 1,
...     'b': [2, Parameter('x'), 5],
...     'c': {1, 2, 3},
... })
>>> d
{'a': 'x+1', 'b': [2, 'x', 5], 'c': {1, 2, 3}}
>>> d.update_parameter('x', 1)
>>> d.eval()
{'a': 2, 'b': [2, 1, 5], 'c': {1, 2, 3}}
```

OrderedParameterDict

class mcalf.utils.collections.**OrderedParameterDict**

Bases: *BaseParameterDict*, OrderedDict

An ordered dictionary of *Parameter* objects.

The same parameters existing across multiple dictionary values can be kept in sync with each other. For a *Parameter* object to be kept in sync it must be located in the dictionary such that `{key: Parameter}` or `{key: List[Parameter]}`.

Examples

```
>>> d = OrderedParameterDict([
...     ('a', Parameter('x') + 1),
...     ('b', [2, Parameter('x'), 5]),
...     ('c', {1, 2, 3}),
... ])
>>> d
OrderedParameterDict([('a', 'x+1'), ('b', [2, 'x', 5]), ('c', {1, 2, 3})])
>>> d.update_parameter('x', 1)
>>> d.eval()
OrderedDict([('a', 2), ('b', [2, 1, 5]), ('c', {1, 2, 3})])
```

BaseParameterDict

class mcalf.utils.collections.**BaseParameterDict**

Bases: *SyncedParameters*

A base class for dictionaries of *Parameter* objects.

The same parameters existing across multiple dictionary values can be kept in sync with each other. For a *Parameter* object to be kept in sync it must be located in the dictionary such that `{key: Parameter}` or `{key: List[Parameter]}`.

Methods Summary

eval()	Return a copy of the dictionary with all *Parameter* objects evaluated.

Methods Documentation

eval()

Return a copy of the dictionary with all *Parameter* objects evaluated.

SyncedParameters

class mcalf.utils.collections.**SyncedParameters**

Bases: object

Database for keeping *Parameter* objects in sync.

Parameter objects with the same name can be linked with this class and kept in sync with each other.

Examples

```
>>> a = Parameter('x') + 1
>>> b = Parameter('x', 2) * 3
>>> c = Parameter('y')
```

Now that *Parameter* objects have been created, we can keep them in sync.

```
>>> s = SyncedParameters()
>>> s.track_object(a)
>>> s.track_object(b)
>>> s.track_object(c)
```

Notice how the value of *a* has been synced to match the value provided in *b*.

```
>>> a == 3
True
```

The value of a named parameter can be updated and the change is propagated to all *Parameter* objects of that name.

```
>>> s.update_parameter('x', 3)
>>> a == 4
True
>>> b == 9
True
```

Methods Summary

exists(name)	Whether any parameters of a particular name are tracked.
get_parameter(name)	Get the value of the named parameter.
has_value(name)	Whether the named parameter has a value set.
track_object(obj)	Add a new *Parameter* object to keep in sync.
update_parameter(name, value)	Update the value of a named parameter.

Methods Documentation

exists(*name: str*) → bool

Whether any parameters of a particular name are tracked.

get_parameter(*name*)

Get the value of the named parameter.

Parameters
name (*str*) – Name of parameter to get value of.

Returns
The current value of the parameter. `None` if not set.

Return type
value

has_value(*name: str*) → bool

Whether the named parameter has a value set.

track_object(*obj*)

Add a new *Parameter* object to keep in sync.

Parameters

obj (*Parameter*) – The parameter object to keep in sync.

Notes

Any objects added must not have a value that contradicts the existing value of the parameter (if set). Objects added which have no value set will inherit the value of the other parameters of the same name.

update_parameter(*name*, *value*)

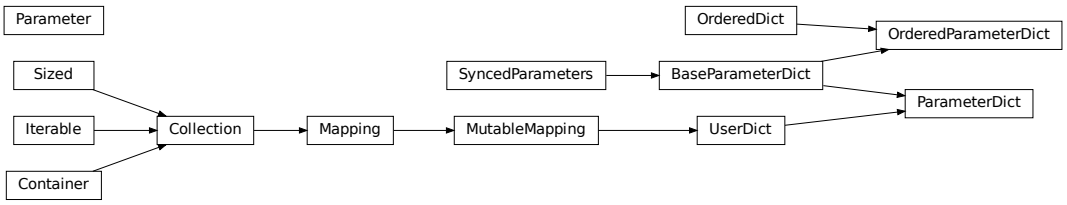
Update the value of a named parameter.

All tracked *Parameter* objects of this name will be updated.

Parameters

- **name** (*string*) – Name of parameter to update.
- **value** (*float or int*) – New value of parameter. Other types may work.

Class Inheritance Diagram



mcalf.utils.misc Module

Functions

make_iter(*args)	Returns each inputted argument, wrapping in a list if not already iterable.
load_parameter(parameter[, wl])	Load parameters from file, optionally evaluating variables from strings.
merge_results(filenames, output)	Merges files generated by the *mcalf.models.FitResults.save()* method.
update_signature(cls)	Update the signature of a model class.

make_iter

mcalf.utils.misc.**make_iter**(**args*)

Returns each inputted argument, wrapping in a list if not already iterable.

Parameters
***args** – Arguments to make iterable.

Returns
**args* converted to iterables.

Return type
iterables

Examples

```
>>> make_iter(1)
[[1]]
```

```
>>> make_iter(1, 2, 3)
[[1], [2], [3]]
```

```
>>> make_iter(1, [2], 3)
[[1], [2], [3]]
```

It is intended that a list of arguments be passed to the function for conversion:

```
>>> make_iter(*[1, [2], 3])
[[1], [2], [3]]
```

Remember that strings are already iterable!

```
>>> make_iter(*[[1, 2, 3], (4, 5, 6), "a"])
[[1, 2, 3], (4, 5, 6), 'a']
```

load_parameter

mcalf.utils.misc.**load_parameter**(*parameter*, *wl=None*)

Load parameters from file, optionally evaluating variables from strings.

Loads the parameter from string or file.

Parameters

- **parameter** (*str*) – Parameter to load, either string of Python list/number or filename string. Supported filename extensions are '.fits', '.fit', '.fts', '.csv', '.txt', '.npy', '.npz', and '.sav'. If the file does not exist, it will assume the string is a Python expression.
- **wl** (*float, optional, default=None*) – Central line core wavelength to replace 'wl' in strings. Will only replace occurrences in the *parameter* variable itself or in files with extension ".csv" or ".txt". When using *wl*, also use 'inf' and 'nan' as required.

Returns
value – Value of parameter in easily computable format (not string).

Return type
numpy.ndarray or list of floats

Examples

```
>>> load_parameter("wl + 4.2", wl=7.1)
11.3
```

```
>>> load_parameter("[wl + 4.2, 5.2 - inf, 5 > 3]", wl=7.1)
[11.3, -inf, 1.0]
```

Filenames are given as follows:

```
>>> x = load_parameter("datafile.csv", wl=12.4)
```

```
>>> x = load_parameter("datafile.fits")
```

If the file does not exist, the function will assume that the string is a Python expression, possibly leading to an error:

```
>>> load_parameter("nonexistant.csv")
Traceback (most recent call last):
 ...
TypeError: 'NoneType' object is not subscriptable
```

merge_results

mcalf.utils.misc.**merge_results**(*filenames*, *output*)

Merges files generated by the *mcalf.models.FitResults.save()* method.

Parameters

- **filenames** (*list of str, length>1*) – List of FITS files generated by *mcalf.models.FitResults.save()* method.
- **output** (*str*) – Name of FITS file to save merged input files to. Will be clobbered.

Notes

See *mcalf.models.FitResults()* for details on the output FITS file data structure.

update_signature

mcalf.utils.misc.**update_signature**(*cls*)

Update the signature of a model class.

Parameters
cls (*type*) – The model class to set a *cls.__init__.__signature__* for.

Notes

This should be called during import of the model class. This function should be called for every class in the model class hierarchy in order starting from *~mcalf.models.ModelBase*.

1.3 Contributor Covenant Code of Conduct

1.3.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

1.3.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

1.3.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

1.3.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

1.3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at mcalf@macbride.me. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

1.3.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

1.3.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

1.4 MCALF Licence

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

P

Q

R

S

T

U

V