csdmpy

A python library for
Core Scientific Dataset Model

Documentation

Release 0.1.5

Deepansh J. Srivastava

Jun 09, 2020

TABLE OF CONTENTS

The *csdmpy* package is the Python support for the core scientific dataset (CSD) model file exchange-format. The package is based on the core scientific dataset (CSD) model, which is designed as a building block in the development of a more sophisticated portable scientific dataset file standard. The CSD model is capable of handling a wide variety of scientific datasets both within and across disciplinary fields.

The main objective of this python package is to facilitate an easy import and export of the CSD model serialized files for Python users. The package utilizes Numpy library and, therefore, offers the end-users versatility to process or visualize the imported datasets with any third-party package(s) compatible with Numpy.

The sample CSDM compliant files used in this documentation are available online.

CHAPTER

ONE

THE CORE SCIENTIFIC DATASET (CSD) MODEL

The core scientific dataset (CSD) model is a *light-weight*, *portable*, *versatile*, and *standalone* data model capable of handling a variety of scientific datasets. The model only encapsulates data values and the minimum metadata to accurately represent a *p*-component dependent variable, $(\mathbf{U}_0, ...\mathbf{U}_q, ...\mathbf{U}_{p-1})$, discretely sampled at *M* unique points in a *d*-dimensional coordinate space, $(\mathbf{X}_0, \mathbf{X}_1, ...\mathbf{X}_k, ...\mathbf{X}_{d-1})$. The model is not intended to encapsulate any information on how the data might be acquired, processed, or visualized.

The data model is *versatile* in allowing many use cases for most spectroscopy, diffraction, and imaging techniques. As such the model supports multi-component datasets associated with continuous physical quantities that are discretely sampled in a multi-dimensional space associated with other carefully controlled quantities, for e.g., a mass as a function of temperature, a current as a function of voltage and time, a signal voltage as a function of magnetic field gradient strength, a color image with a red, green, and blue (RGB) light intensity components as a function of two independent spatial dimensions, or the six components of the symmetric second-rank diffusion tensor MRI as a function of three independent spatial dimensions. Additionally, the model supports multiple dependent variables sharing the same d-dimensional coordinate space. For example, a simultaneous measurement of current and voltage as a function of time, simultaneous acquisition of air temperature, pressure, wind velocity, and solar-flux as a function of Earth's latitude and longitude coordinates. We refer to these dependent variables as *correlated-datasets*.

The CSD model is independent of the hardware, operating system, application software, programming language, and the object-oriented file-serialization format utilized in serializing the CSD model to the file. Out of numerous file serialization formats, XML, JSON, property list, we chose the data-exchange oriented JSON (JavaScript Object Notation) file-serialization format because it is *human-readable* and *easily integrable* with any number of programming languages and field related application-software.

1.1 CSDM

1.1.1 Description

The root level object of the CSD model.

1.1.2 Attributes

Name	Type	Description
version	String	A *required* version number of CSDM file-exchange format.
dimensions	[*Dimension*, …]	A *required* ordered and unique array of dimension objects. An empty array is a valid value.
dependent_variables	[*DependentVariable*, …]	A *required* array of dependent-variable objects. An empty array is a valid value.
tags	[String, …]	An *optional* list of keywords associated with the dataset.
read_only	Boolean	An *optional* value with default as False. If true, the serialized file is archived.
timestamp	String	An *optional* UTC ISO-8601 format timestamp from when the CSDM-compliant file was last serialized.
geographic_coordinate	geographic_coordinate	An *optional* object with attributes required to describe the location from where the CSDM-compliant file was last serialized.
description	String	An *optional* description of the datasets in the CSD model.
application	Generic	An *optional* generic dictionary object containing application specific metadata describing the CSDM object.

1.2 Dimension

A generalized object describing a dimension of a multi-dimensional grid/space.

1.2.1 Attributes

Name	Type	Description
type	*DimObjectSubtype*	A *required* enumeration literal with a valid dimension subtype.
label	String	An *optional* label of the dimension.
description	String	An *optional* description of the dimension.
application	Generic	An *optional* generic dictionary object containing application specific metadata describing the dimension.

1.2.2 Specialized Class

LinearDimension

Description

A LinearDimension is where the coordinates along the dimension follow a linear relationship with the indexes, $\mathbf{J}_k$, along the dimension. Let Δx_k be the *increment*, $N_k \geq 1$, the number of points (*counts*), b_k, the *coordinates offset*, and o_k, the *origin offset* along the k^{th} dimension, then the corresponding coordinates along the dimension, $\mathbf{X}_k$, are given as

$$\mathbf{X}_k = \Delta x_k(\mathbf{J}_k - Z_k) + b_k\mathbf{1},$$

and the absolute coordinates as,

$$\mathbf{X}_k^{\text{abs}} = \mathbf{X}_k + o_k\mathbf{1}.$$

Here, $\mathbf{1}$ is an array of ones, and $\mathbf{J}_k$ is the array of indexes along the k^{th} dimension given as

$$\mathbf{J}_k = [0, 1, 2, 3, ..., N_k - 1].$$

The term, Z_k, is an integer with a value of $Z_k = 0$ or $\frac{T_k}{2}$ when the value of *complex_fft* attribute of the corresponding dimension object is false or true, respectively. Here, $T_k = N_k$ and $N_k - 1$ for even and odd value of N_k, respectively.

Note: When the value of the *complex_fft* attribute is true, and N_k is even, the dependent variable value corresponding to the index $\pm N_k/2$ is an alias.

Attributes

Name	Type	Description
count	Integer	A *required* number of points, N_k, along the dimension.
increment	*ScalarQuantity*	A *required* increment, Δx_k, along the dimension.
coordinates_offset	*ScalarQuantity*	An *optional* coordinate, b_k, corresponding to the zero of the indexes array, $\mathbf{J_k}$. The default value is a physical quantity with zero numerical value.
origin_offset	*ScalarQuantity*	An *optional* origin offset, o_k, along the dimension. The default value is a physical quantity with zero numerical value.
quantity_name	String	An *optional* quantity name associated with the physical quantities describing the dimension.
period	*ScalarQuantity*	An *optional* period of the dimension. By default, the dimension is considered non-periodic.
complex_fft	Boolean	An *optional* boolean with default value as False. If true, the coordinates along the dimension are evaluated as the output of a complex fast Fourier transform (FFT) routine. See the description.
reciprocal	ReciprocalDimension	An *optional* object with attributes required to describe the reciprocal dimension.

Example

The following LinearDimension object,

```
{
    "type": "linear",
    "count": 10,
    "increment": "2 µA",
    "coordinates_offset": "0.1 µA"
}
```

will generate a dimension, where the coordinates $\mathbf{X}_k$ are

```
[
    "0.1 µA",
    "2.1 µA",
    "4.1 µA",
    "6.1 µA",
    "8.1 µA",
    "10.1 µA",
    "12.1 µA",
    "14.1 µA",
    "16.1 µA",
    "18.1 µA"
]
```

MonotonicDimension

Description

A monotonic dimension is a quantitative dimension where the coordinates along the dimension are explicitly defined and, unlike a LinearDimension, may not be derivable from the ordered array of indexes along the dimension. Let $\mathbf{A}_k$ be an ordered set of strictly ascending or descending physical quantities and, o_k, the origin offset along the k^{th} dimension, then the coordinates, $\mathbf{X}_k$, and the absolute coordinates, $\mathbf{X}_k^{\text{abs}}$, along a monotonic dimension follow

$$\mathbf{X}_k = \mathbf{A}_k \text{ and}$$
$$\mathbf{X}_k^{\text{abs}} = \mathbf{X}_k + o_k\mathbf{1},$$

respectively, where $\mathbf{1}$ is an array of ones.

Attributes

Name	Type	Description
coordinates	[*ScalarQuantity*, *ScalarQuantity*, …]	A *required* array of strictly ascending or descending ScalarQuantity.
origin_offset	*ScalarQuantity*	An *optional* origin offset, o_k, along the dimension. The default value is a physical quantity with zero numerical value.
quantity_name	String	An *optional* quantity name associated with the physical quantities describing the dimension.
period	*ScalarQuantity*	An *optional* period of the dimension. By default, the dimension is considered non-periodic.
reciprocal	ReciprocalDimension	An *optional* object with attributes required to describe the reciprocal dimension.

Example

The following MonotonicDimension object,

```
{
    "type": "monotonic",
    "coordinates": ["1 µs", "10 µs", "100 µs", "1 ms", "10 ms", "100 ms", "1 s", "10 s
↪"]
}
```

will generate a dimension, where the coordinates $\mathbf{X}_k$ are

```
["1 µs", "10 µs", "100 µs", "1 ms", "10 ms", "100 ms", "1 s", "10 s"]
```

LabeledDimension

Description

A labeled dimension is a qualitative dimension where the coordinates along the dimension are explicitly defined as labels. Let $\mathbf{A}_k$ be an ordered set of unique labels along the k^{th} dimension, then the coordinates, $\mathbf{X}_k$, along a labeled dimension are

$$\mathbf{X}_k = \mathbf{A}_k.$$

Attributes

Name	Type	Description
labels	[String, String, …]	A *required* ordered array of labels along the dimension.

Example

The following LabeledDimension object,

```
{
    "type": "labeled",
    "labels": ["Cu", "Fe", "Si", "H", "Li"]
}
```

will generate a dimension, where the coordinates $\mathbf{X}_k$ are

```
["Cu", "Fe", "Si", "H", "Li"]
```

1.3 DependentVariable

1.3.1 Description

A generalized object describing a dependent variable of the dataset, which holds an ordered list of *p* components, indexed as *q=0* to *p-1*, as

$$[\mathbf{U}_0, ...\mathbf{U}_q, ...\mathbf{U}_{p-1}].$$

1.3.2 Attributes

Name	Type	Description
type	*DVObjectSubtype*	An enumeration literal with a valid dependent variable subtype.
name	String	Name of the dependent variable.
unit	String	The unit associated with the physical quantities describing the dependent variable.
quantity_name	String	Quantity name associated with the physical quantities describing the dependent variable.
numeric_type	*NumericType*	An enumeration literal with a valid numeric type.
quantity_type	*QuantityType*	An enumeration literal with a valid quantity type.
component_labels	[String, String, …]	Ordered array of labels associated with ordered array of components of the dependent variable.
sparse_sampling	sparseSampling_uml	Object with attribute required to describe a sparsely sampled dependent variable components.
description	String	Description of the dependent variable.
application	Generic	Generic dictionary object containing application specific metadata describing the dependent variable.

1.3.3 Specialized Class

InternalDependentVariable

Description

An InternalDependentVariable is where the components of the dependent variable are defined within the object as the value of the *components* key, along with other metadata describing the dependent variable.

Attributes

Name	Type	Description
components	[String, String, …]	A *required* attribute. The value is an array of base64 encoded strings where each string is a component of the dependent variable and decodes to a binary array of *M* data values. This value is only *valid* only when the corresponding value of the *encoding* attribute is *base64*.
components	[[Float, Float, …], [Float, Float, …], …]	A *required* attribute. The value is an array of arrays where each inner array is a component of the dependent variable with *M* data values. This value is *valid* only when the value of *encoding* is *none*.
encoding	String	A required enumeration literal, where the valid literals are *none* or *base64*.

ExternalDependentVariable

Description

An ExternalDependentVariable is where the components of the dependent variable are defined in an external file whose location is defined as the value of the `components_url` key.

Attributes

Name	Type	Description
components_url	String	A *required* URL location where the components of the dependent variable are serialized as a binary data.

SparseSampling

Description

A SparseSampling object describes the dimensions indexes and grid vertexes where the components of the dependent variable are sparsely sampled.

Attributes

Name	Type	Description
dimension_indexes	[Integer, Integer, …]	A *required* array of integers indicating along which dimensions the *DependentVariable* is sparsely sampled.
sparse_grid_vertexes	[Integer, Integer, …]	A required flattened array of integer indexes along the sparse dimensions where the components of the dependent variable are sampled. This is only *valid* when the encoding from the corresponding SparseSampling object is `none`.
sparse_grid_vertexes	String	A *required* base64 string of a flattened binary array of integer indexes along the sparse dimensions where the components of the dependent variable are sampled. This is only *valid* when the encoding from the corresponding SparseSampling object is `base64`.
encoding	String	A *required* enumeration with the following valid literals. `none`, `base64`
unsigned_integer_type	String	A *required* enumeration with the following valid literals. `uint8`, `uint16`, `uint32`, `uint64`
description	String	An *optional* description of the dependent variable.
application	Generic	An *optional* generic dictionary object containing application specific metadata describing the SparseSampling object.

1.4 Enumeration

1.4.1 DimObjectSubtype

An enumeration with literals as the value of the *Dimension* objects' *type* attribute.

Literal	Description
linear	Literal specifying an instance of a linearDimension_uml object.
monotonic	Literal specifying an instance of a monotonicDimension_uml object.
labeled	Literal specifying an instance of a labeledDimension_uml object.

1.4.2 DVObjectSubtype

An enumeration with literals as the values of the *DependentVariable* object' *type* attribute.

Literal	Description
internal	Literal specifying an instance of an internal_uml object.
external	Literal specifying an instance of an external_uml object.

1.4.3 NumericType

An enumeration with literals as the value of the *DependentVariable* objects' *numeric_type* attribute.

Literal	Description
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
float32	32-bit floating point number
float64	64-bit floating point number
complex64	two 32-bit floating points numbers
complex128	two 64-bit floating points numbers

1.4.4 QuantityType

An enumeration with literals as the value of the *DependentVariable* objects' *quantity_type* attribute. The value is used in interpreting the *p*-components of the dependent variable.

- scalar A dependent variable with $p = 1$ component interpret as a scalar, $\mathcal{S}_i = U_{0,i}$.
- vector_n A dependent variable with $p = n$ components interpret as vector components, $\mathcal{V}_i = [U_{0,i}, U_{1,i}, ...U_{n-1,i}]$.
- matrix_n_m A dependent variable with $p = mn$ components interpret as a $n \times m$ matrix as follows,

$$M_i = \begin{bmatrix} U_{0,i} & U_{1,i} & ... & U_{(n-1)m,i} \\ U_{1,i} & U_{m+1,i} & ... & U_{(n-1)m+1,i} \\ \vdots & \vdots & \vdots & \vdots \\ U_{m-1,i} & U_{2m-1,i} & ... & U_{nm-1,i} \end{bmatrix}$$

- symmetric_matrix_n A dependent variable with $p = n^2$ components interpret as a matrix symmetric about its leading diagonal as shown below,

$$M_i^{(s)} = \begin{bmatrix} U_{0,i} & U_{1,i} & ... & U_{n-1,i} \\ U_{1,i} & U_{n,i} & ... & U_{2n-2,i} \\ \vdots & \vdots & \vdots & \vdots \\ U_{n-1,i} & U_{2n-2,i} & ... & U_{\frac{n(n+1)}{2}-1,i} \end{bmatrix}$$

- pixel_n A dependent variable with $p = n$ components interpret as image/pixel components, $\mathcal{P}_i = [U_{0,i}, U_{1,i}, ...U_{n-1,i}]$.

Here, the terms n and m are intergers.

1.5 ScalarQuantity

ScalarQuantity is an object composed of a numerical value and any valid SI unit symbol or any number of accepted non-SI unit symbols. It is serialized in the JSON file as a string containing a numerical value followed by the unit symbol, for example,

- "3.4 m" (SI)
- "2.3 bar" (non-SI)

CHAPTER

TWO

CSDMPY PACKAGE

2.1 Installing *csdmpy* package

Using PIP:

PIP is a package manager for Python packages and is included with python version 3.4 and higher.

```
$ pip install csdmpy
```

2.2 The requirements for *csdmpy*

The list of packages dependencies for *csdmpy* module

- numpy>=1.10.1 (for handling n-dimensional arrays)
- setuptools>=27.3
- requests>=2.21.0 (for downloading files from server)
- astropy>=3.0 (for astropy units module)
- matplotlib>=3.0 (for rendering plots)

CHAPTER

THREE

GETTING STARTED WITH *CSDMPY* PACKAGE

We have put together a set of guidelines for importing the *csdmpy* package and related methods and attributes. We encourage the users to follow these guidelines to promote consistency, amongst others. Import the package using

```
>>> import csdmpy as cp
```

To load a *.csdf* or a *.csdfe* file, use the `load()` method of the *csdmpy* module. In the following example, we load a sample test file.

```
>>> filename = cp.tests.test01 # replace this with your file's name.
>>> testdata1 = cp.load(filename)
```

Here, `testdata1` is an instance of the CSDM class.

At the root level, the *CSDM* object includes various useful optional attributes that may contain additional information about the dataset. One such useful attribute is the `description` key, which briefs the end-users on the contents of the dataset. To access the value of this attribute use,

```
>>> testdata1.description
'A simulated sine curve.'
```

3.1 Accessing the dimensions and dependent variables of the dataset

An instance of the CSDM object may include multiple dimensions and dependent variables. Collectively, the dimensions form a multi-dimensional grid system, and the dependent variables populate this grid. In *csdmpy*, dimensions and dependent variables are structured as python's tuple object. To access these tuples, use the `dimensions` and `dependent_variables` attribute of the CSDM object, respectively. For example,

```
>>> x = testdata1.dimensions
>>> y = testdata1.dependent_variables
```

In this example, the dataset contains one dimension and one dependent variable, and therefore, *x* and *y* are tuples with a single instance.

```
>>> print('x is a {0} of length {1}.'.format(type(x).__name__, len(x)))
x is a tuple of length 1.
>>> print('y is a {0} of length {1}.'.format(type(y).__name__, len(y)))
y is a tuple of length 1.
```

You may access the instances of individual dimension and dependent variable by using the proper indexing. For example, the dimension and dependent variable at index 0 may be accessed using `x[0]` and `y[0]`, respectively.

Every instance of the *Dimension* object has its own set of attributes that further describe the respective dimension. For example, a Dimension object may have an optional `description` attribute,

```
>>> x[0].description
'A temporal dimension.'
```

Similarly, every instance of the *DependentVariable* object has its own set of attributes. In this example, the `description` attribute from the dependent variable is

```
>>> y[0].description
'A response dependent variable.'
```

3.1.1 Coordinates along the dimension

Every dimension object contains a list of coordinates associated with every grid index along the dimension. To access these coordinates, use the `coordinates` attribute of the respective *Dimension* instance. In this example, the coordinates are

```
>>> x[0].coordinates
<Quantity [0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] s>
```

Note: `x[0].coordinates` returns a Quantity instance from the Astropy package. The *csdmpy* module utilizes the units library from astropy.units module to handle physical quantities. The numerical *value* and the *unit* of the physical quantities are accessed through the Quantity instance, using the `value` and the `unit` attributes, respectively. Please refer to the astropy.units documentation for details. In the *csdmpy* module, the `Quantity.value` is a Numpy array. For instance, in the above example, the underlying Numpy array from the coordinates attribute is accessed as

```
>>> x[0].coordinates.value
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

3.1.2 Components of the dependent variable

Every dependent variable object has at least one component. The number of components of the dependent variable is determined from the `quantity_type` attribute of the dependent variable object. For example, a scalar quantity has one-component, while a vector quantity may have multiple components. To access the components of the dependent variable, use the `components` attribute of the respective *DependentVariable* instance. For example,

```
>>> y[0].components
array([[ 0.0000000e+00,  5.8778524e-01,  9.5105654e-01,  9.5105654e-01,
         5.8778524e-01,  1.2246469e-16, -5.8778524e-01, -9.5105654e-01,
        -9.5105654e-01, -5.8778524e-01]], dtype=float32)
```

The `components` attribute is a Numpy array. Note, the number of dimensions of this array is $d + 1$, where d is the number of *Dimension* objects from the `dimensions` attribute. The additional dimension in the Numpy array corresponds to the number of components of the dependent variable. For instance, in this example, there is a single dimension, *i.e.*, $d = 1$ and, therefore, the value of the `components` attribute holds a two-dimensional Numpy array of shape

```
>>> y[0].components.shape
(1, 10)
```

where the first element of the shape tuple, *1*, is the number of components of the dependent variable and the second element, *10*, is the number of points along the dimension, *i.e.*, `x[0].coordinates`.

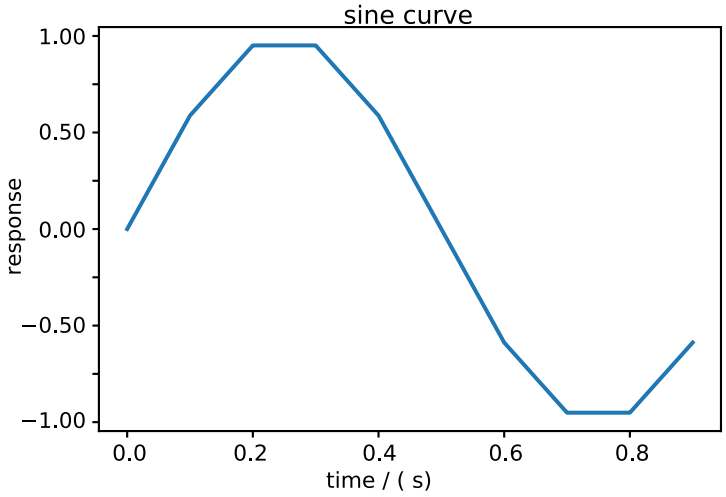
3.2 Plotting the dataset

It is always helpful to represent a scientific dataset with visual aids such as a plot or a figure instead of columns of numbers. As such, throughout this documentation, we provide a figure or two for every example dataset. We make use of Python's Matplotlib library for generating these figures. The users may, however, use their favorite plotting library.

Attention: Although we show code for visualizing the dataset, this documentation is not a guide for data visualization.

The following snippet plots the dataset from this example. Here, the *axis_label* is an attribute of both Dimension and DependentVariable instances, and the *name* is an attribute of the DependentVariable instance.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x[0].coordinates, y[0].components[0])
>>> plt.xlabel(x[0].axis_label)
>>> plt.ylabel(y[0].axis_label[0])
>>> plt.title(y[0].name)
>>> plt.show()
```



See also:

CSDM, *Dimension*, *DependentVariable*, Quantity, numpy array, Matplotlib library

CHAPTER

FOUR

EXAMPLES

In this section, we present illustrative examples for importing files serialized with the CSD model using the *csdmpy* package. Because the CSD model allows multi-dimensional datasets with multiple dependent variables, we use a shorthand notation of $d\text{D}\{p\}$ to indicate that a dataset has a p-component dependent variable defined on a d-dimensional coordinate grid. In the case of *correlated datasets*, the number of components in each dependent variable is given as a list within the curly braces, *i.e.*, $d\text{D}\{p_0, p_1, p_2, ...\}$.

The sample CSDM compliant files used in this documentation are available online.

Example Dataset

4.1 Scalar, 1D{1} datasets

The 1D{1} datasets are one dimensional, $d = 1$, with one single-component, $p = 1$, dependent variable. In this section, we walk through some examples of 1D{1} datasets.

Let's start by first importing the *csdmpy* module.

```
>>> import csdmpy as cp
```

4.1.1 Global Mean Sea Level rise dataset

The following dataset is the Global Mean Sea Level (GMSL) rise from the late 19th to the Early 21st Century. The original dataset was downloaded as a CSV file and subsequently converted to the CSD model format. Let's import this file.

```
>>> filename = 'Test Files/gmsl/GMSL.csdf'
>>> sea_level = cp.load(filename)
```

The variable *filename* is a string with the address to the *sea_level.csdf* file. The `load()` method of the *csdmpy* module reads the file and returns an instance of the *CSDM* class, in this case, as a variable `sea_level`. For a quick preview of the data structure, use the `data_structure` attribute of this instance.

```
>>> print(sea_level.data_structure)
{
  "csdm": {
    "version": "1.0",
```

(continues on next page)

(continued from previous page)


```
    "read_only": true,
    "timestamp": "2019-05-21T13:43:00Z",
    "tags": [
      "Jason-2",
      "satellite altimetry",
      "mean sea level",
      "climate"
    ],
    "description": "Global Mean Sea Level (GMSL) rise from the late 19th to the Early␣
↪21st Century.",
    "dimensions": [
      {
        "type": "linear",
        "count": 1608,
        "increment": "0.08333333333 yr",
        "coordinates_offset": "1880.0416666667 yr",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Global Mean Sea Level",
        "unit": "mm",
        "quantity_name": "length",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "GMSL"
        ],
        "components": [
          [
            "-183.0, -171.125, ..., 59.6875, 58.5"
          ]
        ]
      }
    ]
  }
}
```


Warning: The serialized string from the *data_structure* attribute is not the same as the JSON serialization on the file. This attribute is only intended for a quick preview of the data structure and avoids displaying large datasets. Do not use the value of this attribute to save the data to the file. Instead, use the *save()* method of the *CSDM* class.

The tuples of the dimensions and dependent variables from this example are

```
>>> x = sea_level.dimensions
>>> y = sea_level.dependent_variables
```

respectively. The coordinates along the dimension and the component of the dependent variable are

```
>>> print(x[0].coordinates)
[1880.04166667 1880.125      1880.20833333 ... 2013.79166666 2013.87499999
 2013.95833333] yr

>>> print(y[0].components[0])
[-183.     -171.125  -164.25   ...   66.375    59.6875   58.5    ]
```

respectively.

Tip: Plotting a one-dimension scalar line-plot.

Before we plot this dataset, we find it convenient to write a small plotting method. This method makes it easier, later, when we describe 1D{1} examples form a variety of scientific datasets. The method follows-

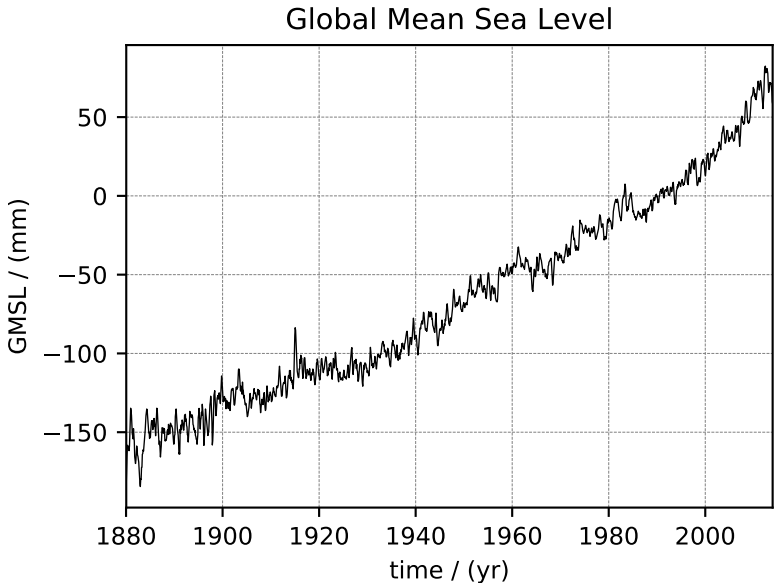
```
>>> import matplotlib.pyplot as plt
>>> def plot1D(dataObject):
...     # tuples of dependent and dimension instances.
...     x = dataObject.dimensions
...     y = dataObject.dependent_variables
...
...     plt.figure(figsize=(4,3))
...     plt.plot(x[0].coordinates, y[0].components[0].real, color='k', linewidth=0.5)
...
...     plt.xlim(x[0].coordinates[0].value, x[0].coordinates[-1].value)
...
...     # The axes labels and figure title.
...     plt.xlabel(x[0].axis_label)
...     plt.ylabel(y[0].axis_label[0])
...     plt.title(y[0].name)
...
...     plt.grid(color='gray', linestyle='--', linewidth=0.3)
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

A quick walk-through of the `plot1D` method. The method accepts an instance of the *CSDM* class as an argument. Within the method, we make use of the instance's attributes in addition to the matplotlib functions. The first line assigns the tuple of the dimensions and dependent variables to `x` and `y`, respectively. The following two lines add a plot of the components of the dependent variable versus the coordinates of the dimension. The next line sets the x-range. For labeling the axes, we use the *axis_label* attribute of both dimension and dependent variable instances. For the figure title, we use the *name* attribute of the dependent variable instance. The next statement adds the grid lines. For additional information, refer to Matplotlib documentation.

The `plot1D` method is only for illustrative purposes. The users may use any plotting library to visualize their datasets.

Now to plot the *sea_level* dataset.

```
>>> plot1D(sea_level)
```



4.1.2 Nuclear Magnetic Resonance (NMR) dataset

The following dataset is a ^{13}C time-domain NMR Bloch decay signal of ethanol. Let's load this data file and take a quick look at its data structure. We follow the previously described steps.

```
>>> filename = 'Test Files/NMR/blochDecay/blochDecay.csdf'
>>> NMR_data = cp.load(filename)
>>> print(NMR_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2016-03-12T16:41:00Z",
    "geographic_coordinate": {
      "altitude": "238.9719543457031 m",
      "longitude": "-83.05154573892345 °",
      "latitude": "39.97968794964322 °"
    },
    "tags": [
      "13C",
      "NMR",
      "spectrum",
      "ethanol"
    ],
    "description": "A time domain NMR 13C Bloch decay signal of ethanol.",
    "dimensions": [
      {
        "type": "linear",
        "count": 4096,
        "increment": "0.1 ms",
        "coordinates_offset": "-0.3 ms",
        "quantity_name": "time",
        "reciprocal": {
          "coordinates_offset": "-3005.363 Hz",
          "origin_offset": "75426328.86 Hz",
          "quantity_name": "frequency",
```

(continues on next page)

(continued from previous page)

```
          "label": "13C frequency shift"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "numeric_type": "complex128",
        "quantity_type": "scalar",
        "components": [
          [
            "(-8899.40625-1276.7734375j), (-4606.88037109375-742.41247558593375j), ...,
↪ (37.548492431640625+20.156890869140625j), (-193.9228515625-67.06524658203125j)"
          ]
        ]
      }
    ]
  }
}
```

This particular example illustrates two additional attributes of the CSD model, namely, the *geographic_coordinate* and *tags*. The *geographic_coordinate* described the location where the CSDM file was last serialized. You may access this attribute through,

```
>>> NMR_data.geographic_coordinate
{'altitude': '238.9719543457031 m', 'longitude': '-83.05154573892345 °', 'latitude':
↪'39.97968794964322 °'}
```

Similarly, the *tags* attribute can be accessed through,

```
>>> NMR_data.tags
['13C', 'NMR', 'spectrum', 'ethanol']
```

You may add additional tags, if so desired, to this list using the *append* method of python's list class, such as

```
>>> NMR_data.tags.append("Bloch decay")
>>> NMR_data.tags
['13C', 'NMR', 'spectrum', 'ethanol', 'Bloch decay']
```

Unlike the previous example, the data structure of the NMR measurement is a complex-valued dependent variable where the values are

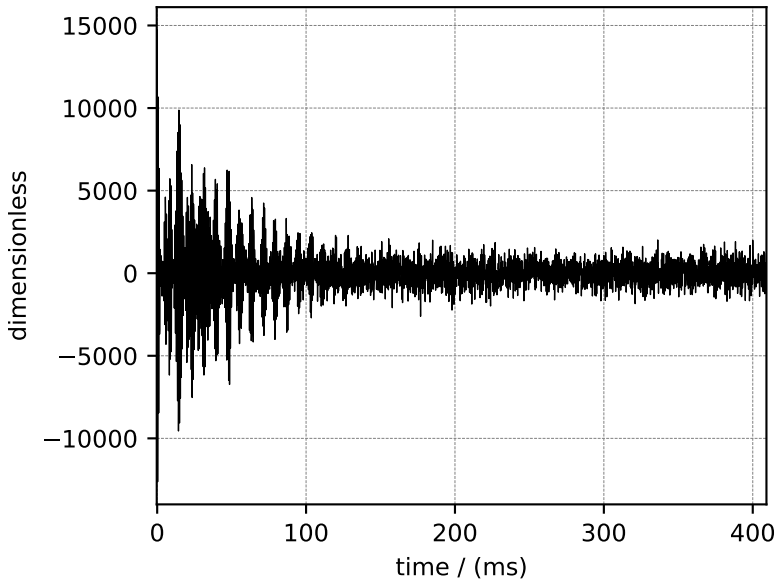
```
>>> y = NMR_data.dependent_variables
>>> print(y[0].components[0])
[-8899.40625   -1276.7734375j  -4606.88037109 -742.41247559j
  9486.43847656 -770.0413208j  ...   -70.95385742  -28.32843018j
    37.54849243  +20.15689087j  -193.92285156  -67.06524658j]
```

The coordinates along the dimension are

```
>>> x = NMR_data.dimensions
>>> x0 = x[0].coordinates
>>> print(x0)
[-3.000e-01 -2.000e-01 -1.000e-01 ...  4.090e+02  4.091e+02  4.092e+02] ms
```

Now to the plot the dataset,

```
>>> plot1D(NMR_data)
```



4.1.3 Electron Paramagnetic Resonance (EPR) dataset

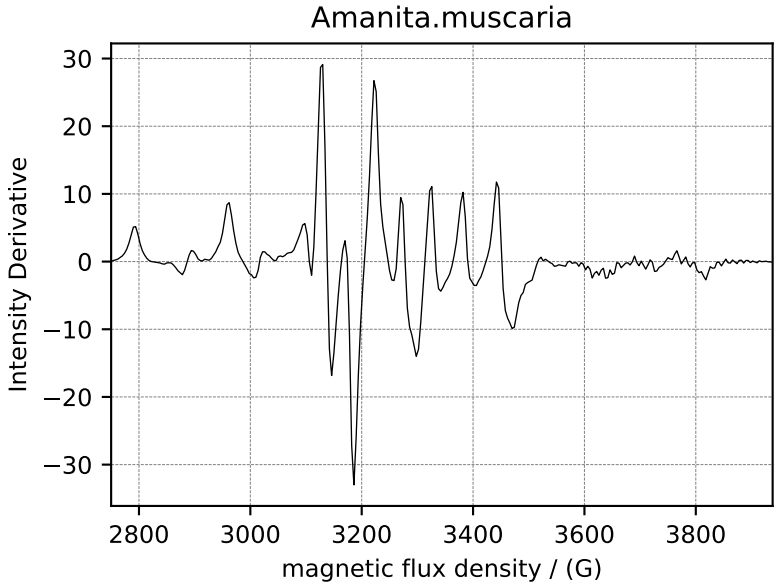
The following simulation of the EPR dataset is formerly obtained as a JCAMP-DX file and subsequently converted to the CSD model file-format. The data structure of this dataset and the corresponding plot follows,

```
>>> filename = 'Test Files/EPR/AmanitaMuscaria_base64.csdf'
>>> EPR_data = cp.load(filename)
>>> print(EPR_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2015-02-26T16:41:00Z",
    "description": "A Electron Paramagnetic Resonance simulated dataset.",
    "dimensions": [
      {
        "type": "linear",
        "count": 298,
        "increment": "4.0 G",
        "coordinates_offset": "2750.0 G",
        "quantity_name": "magnetic flux density"
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Amanita.muscaria",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "Intensity Derivative"
        ],
```

(continues on next page)

(continued from previous page)

```
        "components": [
          [
            "0.067, 0.136, ..., -0.035, -0.137"
          ]
        ]
      }
    ]
  }
}
>>> plot1D(EPR_data)
```



4.1.4 Gas Chromatography dataset

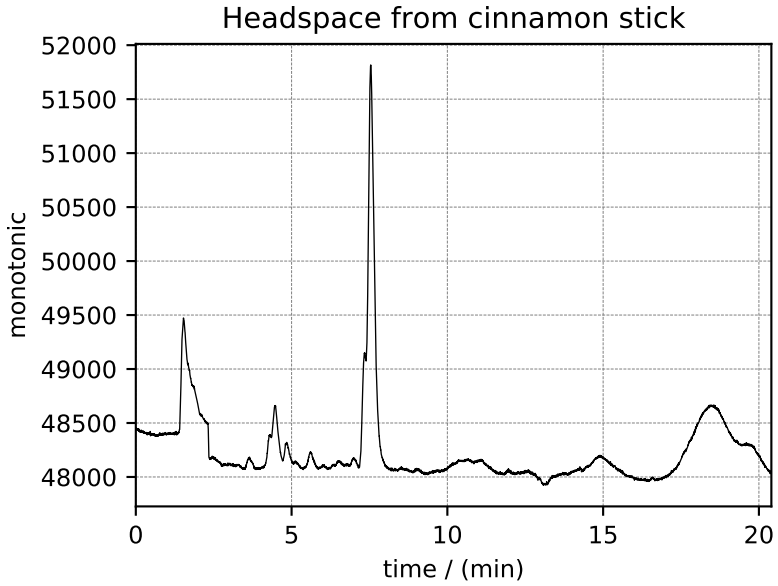
The following Gas Chromatography dataset is also obtained as a JCAMP-DX file and subsequently converted to the CSD model file-format. The data structure and the plot of the gas chromatography dataset follows,

```
>>> filename = 'Test Files/GC/cinnamon_none.csdf'
>>> GCData = cp.load(filename)
>>> print(GCData.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2011-12-16T12:24:10Z",
    "description": "A Gas Chromatography dataset of cinnamon stick.",
    "dimensions": [
      {
        "type": "linear",
        "count": 6001,
        "increment": "0.0034 min",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
```

(continues on next page)

(continued from previous page)

```
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Headspace from cinnamon stick",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "monotonic"
        ],
        "components": [
          [
            "48453.0, 48444.0, ..., 48040.0, 48040.0"
          ]
        ]
      }
    ]
  }
}
>>> plot1D(GCData)
```



4.1.5 Fourier Transform Infrared Spectroscopy (FTIR) dataset

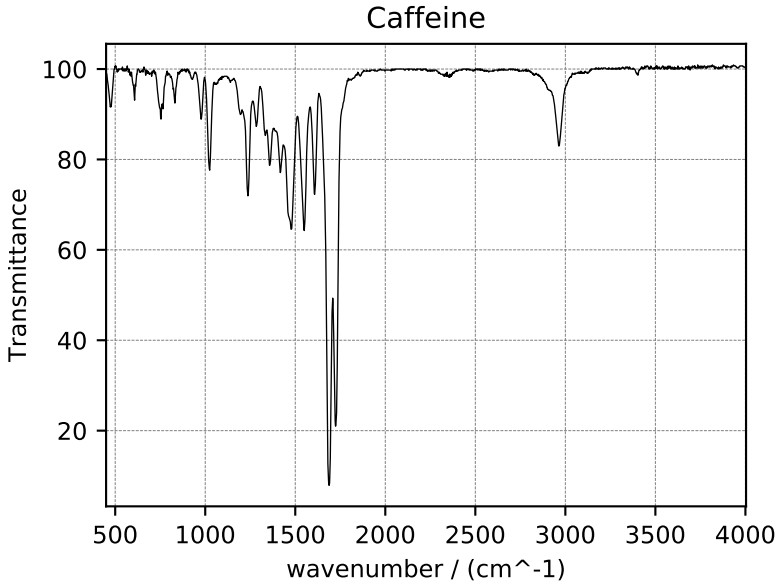
For the following FTIR dataset, we again convert the original JCAMP-DX file to the CSD model file-format. The data structure and the plot of the FTIR dataset follows,

```
>>> filename = 'Test Files/IR/caffeine_none.csdf'
>>> FTIR_data = cp.load(filename)
>>> print(FTIR_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
```

(continues on next page)

(continued from previous page)

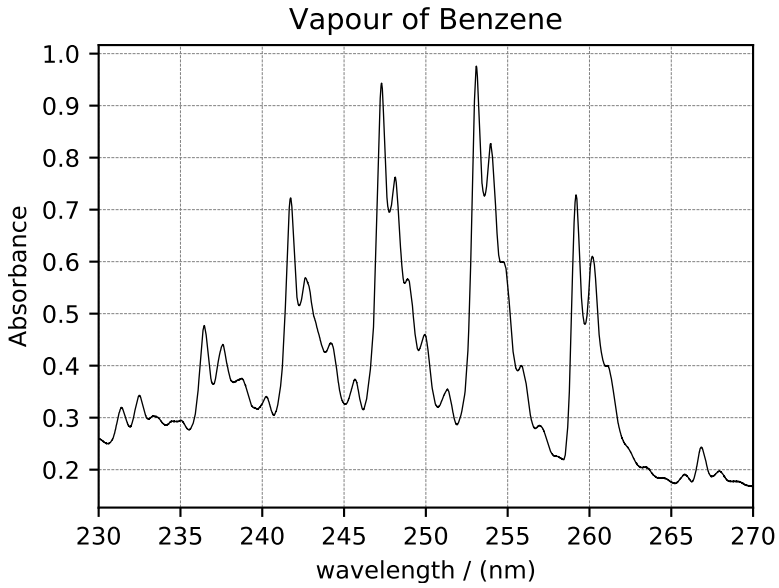
```
    "timestamp": "2019-07-01T21:03:42Z",
    "description": "An IR spectrum of caffeine.",
    "dimensions": [
      {
        "type": "linear",
        "count": 1842,
        "increment": "1.930548614883216 cm^-1",
        "coordinates_offset": "449.41 cm^-1",
        "quantity_name": "wavenumber",
        "reciprocal": {
          "quantity_name": "length"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Caffeine",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "Transmittance"
        ],
        "components": [
          [
            "99.31053, 99.08212, ..., 100.22944, 100.22944"
          ]
        ]
      }
    ]
  }
}
>>> plot1D(FTIR_data)
```



4.1.6 Ultraviolet–visible (UV-vis) dataset

The following UV-vis dataset is originally downloaded as a JCAMP-DX file and consequently turned to the CSD model file-format. The data structure and the plot of the UV-vis dataset follows,

```
>>> filename = 'Test Files/UV-Vis/benzeneVapour_base64.csdf'
>>> UV_data = cp.load(filename)
>>> print(UV_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2014-09-30T11:16:33Z",
    "description": "A UV-vis spectra of benzene vapours.",
    "dimensions": [
      {
        "type": "linear",
        "count": 4001,
        "increment": "0.01 nm",
        "coordinates_offset": "230.0 nm",
        "quantity_name": "length",
        "label": "wavelength",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Vapour of Benzene",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "Absorbance"
        ],
        "components": [
          [
            "0.25890622, 0.25923702, ..., 0.16814752, 0.16786034"
          ]
        ]
      }
    ]
  }
}
>>> plot1D(UV_data)
```



4.1.7 Mass spectrometry dataset

The following is an example of a sparse dataset. The *acetone.csdf* CSDM data file is stored as a sparse dependent variable data. Upon import, the values of the dependent variable component sparsely populate the coordinate grid. The remaining unpopulated coordinates are assigned a zero value.

```
>>> filename = 'Test Files/MassSpec/acetone.csdf'
>>> mass_spec = cp.load(filename)
>>> print(mass_spec.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2019-06-23T17:53:26Z",
    "description": "MASS spectrum of acetone",
    "dimensions": [
      {
        "type": "linear",
        "count": 51,
        "increment": "1.0",
        "coordinates_offset": "10.0",
        "label": "m/z"
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "acetone",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "component_labels": [
          "relative abundance"
        ],
        "components": [
          [
            "0.0, 0.0, ..., 10.0, 0.0"
```

(continues on next page)

(continued from previous page)

```
            ]
          ]
        }
      ]
    }
}
```

Here, the coordinates along the dimension are

```
>>> print(mass_spec.dimensions[0].coordinates)
[10. 11. 12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27.
 28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43. 44. 45.
 46. 47. 48. 49. 50. 51. 52. 53. 54. 55. 56. 57. 58. 59. 60.]
```

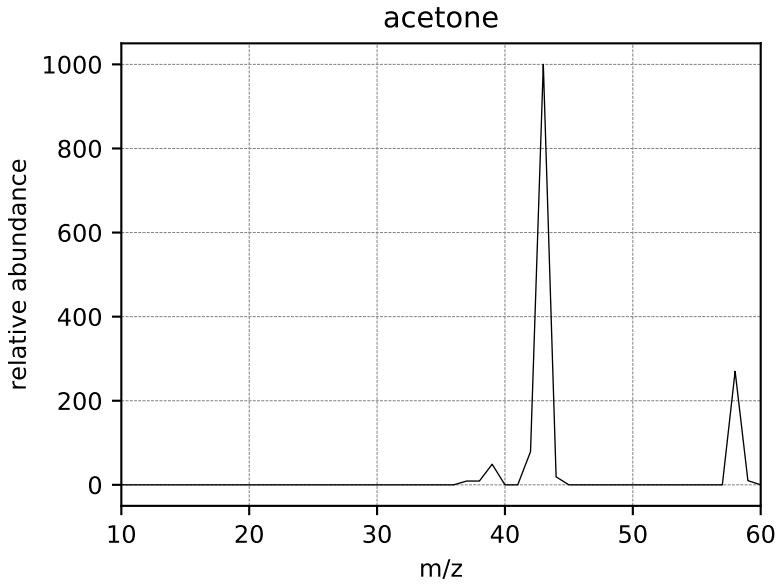
and the components of the dependent variable follow

```
>>> print(mass_spec.dependent_variables[0].components[0])
[   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
    0.    0.    0.    9.    9.   49.    0.    0.   79. 1000.   19.    0.
    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
  270.   10.    0.]
```

Note, only eight values were specified in the dependent variable *components* attribute in the *.csdf* file. The remaining component values are set to zero.

Now to plot the dataset.

```
>>> plot1D(mass_spec)
```



4.2 Scalar, 2D{1} datasets

The 2D{1} datasets are two dimensional, $d = 2$, with one single-component dependent variable, $p = 1$. Here, we present examples of 2D{1} datasets with the scalar dependent variable.

4.2.1 Astronomy dataset

The following dataset is a new observation of the Bubble Nebula acquired by The Hubble Heritage Team, in February 2016. The original dataset was obtained in the FITS format and subsequently converted to the CSD model file-format. For the convenience of illustration, we have downsampled the original dataset.

Let's load the *.csdfe* file and look at its data structure.

```
>>> import csdmpy as cp
>>> bubble_nebula = cp.load('Test Files/BubbleNebula/Bubble.csdfe')
>>> print(bubble_nebula.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2016-02-26T16:41:00Z",
    "description": "The dataset is a new observation of the Bubble Nebula acquired by␣
↪The Hubble Heritage Team, in February 2016.",
    "dimensions": [
      {
        "type": "linear",
        "count": 1024,
        "increment": "-0.0002581136196 °",
        "coordinates_offset": "350.311874957 °",
        "quantity_name": "plane angle",
        "label": "Right Ascension"
      },
      {
        "type": "linear",
        "count": 1024,
        "increment": "0.0001219957797701109 °",
        "coordinates_offset": "61.12851494969163 °",
        "quantity_name": "plane angle",
        "label": "Declination"
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Bubble Nebula, 656nm",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "components": [
          [
            "0.0, 0.0, ..., 0.0, 0.0"
          ]
        ]
      }
    ]
  }
}
```

Here, the variable `bubble_nebula` is an instance of the *CSDM* class. From the data structure, one finds two dimensions, labeled as *Right Ascension* and *Declination*, and one single-component dependent variable named *Bubble Nebula, 656nm*.

Let's get the tuples of the dimension and dependent variable instances from the `bubble_nebula` instance following,

```
>>> x = bubble_nebula.dimensions
>>> y = bubble_nebula.dependent_variables
```

There are two dimension instances in `x`. Let's look at the coordinates along each dimension, using the *coordinates* attribute of the respective instances.

```
>>> print(x[0].coordinates[:10])
[350.31187496 350.31161684 350.31135873 350.31110062 350.3108425
 350.31058439 350.31032628 350.31006816 350.30981005 350.30955193] deg

>>> print(x[1].coordinates[:10])
[61.12851495 61.12863695 61.12875894 61.12888094 61.12900293 61.12912493
 61.12924692 61.12936892 61.12949092 61.12961291] deg
```

Here, we only print the first ten coordinates along the respective dimensions.

The component of the dependent variable is accessed through the *components* attribute.

```
>>> y00 = y[0].components[0]
```

Visualizing the dataset

Now, to plot the dataset.

Tip: Intensity plot.

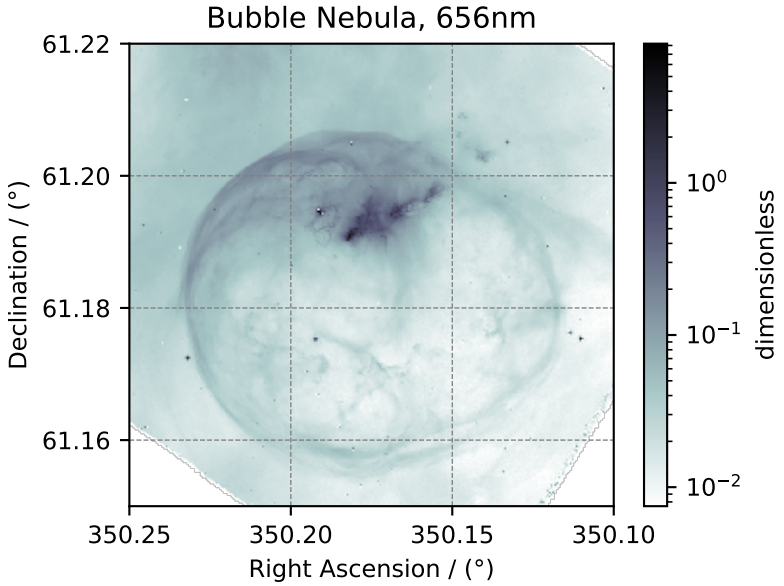
```
>>> import matplotlib.pyplot as plt
>>> from matplotlib.colors import LogNorm
>>> import numpy as np

>>> def plot():
...     # Figure setup.
...     fig, ax = plt.subplots(1,1, figsize=(4,3))
...     ax.set_facecolor('w')
...
...     # the coordinates along the two dimensions
...     x0 = x[0].coordinates
...     x1 = x[1].coordinates
...
...     # Set the extents of the image.
...     extent=[x0[0].value, x0[-1].value, x1[0].value, x1[-1].value]
...
...     # Log intensity image plot.
...     im = ax.imshow(np.abs(y00), origin='lower', cmap='bone_r',
...                    norm=LogNorm(vmax=y00.max()/10, vmin=7.5e-3, clip=True),
...                    extent=extent, aspect='auto')
...
...     # Set the axes labels and the figure tile.
...     ax.set_xlabel(x[0].axis_label)
...     ax.set_ylabel(x[1].axis_label)
...     ax.set_title(y[0].name)
...     ax.locator_params(nbins=5)
```

(continues on next page)

(continued from previous page)

```
...
...     # Add a colorbar.
...     cbar = fig.colorbar(im)
...     cbar.ax.set_ylabel(y[0].axis_label[0])
...
...     # Set the x and y limits.
...     ax.set_xlim([350.25, 350.1])
...     ax.set_ylim([61.15, 61.22])
...
...     # Add grid lines.
...     ax.grid(color='gray', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot()
```



4.2.2 Nuclear Magnetic Resonance (NMR) dataset

The following example is a ^{29}Si NMR time-domain saturation recovery measurement of a highly siliceous zeolite ZSM-12. Usually, the spin recovery measurements are acquired over a rectilinear grid where measurements along one of the dimensions are non-uniform and span several orders of magnitude. In this example, we illustrate the use of *monotonic* dimensions for describing such datasets.

Let's load the file.

```
>>> import csdmpy as cp

>>> filename = 'Test Files/NMR/satrec/satRec.csdf'
>>> NMR_2D_data = cp.load(filename)
```

The tuples of the dimension and dependent variable instances from the `NMR_2D_data` instance are

```
>>> x = NMR_2D_data.dimensions
>>> y = NMR_2D_data.dependent_variables
```

respectively. There are two dimension instances in this example with respective dimension data structures as

```
>>> print(x[0].data_structure)
{
  "type": "linear",
  "description": "A full echo echo acquisition along the t2 dimension using a Hahn␣
↪echo.",
  "count": 1024,
  "increment": "80.0 µs",
  "coordinates_offset": "-41.04 ms",
  "quantity_name": "time",
  "label": "t2",
  "reciprocal": {
    "coordinates_offset": "-8766.0626 Hz",
    "origin_offset": "79578822.26200001 Hz",
    "quantity_name": "frequency",
    "label": "29Si frequency shift"
  }
}
```

and

```
>>> print(x[1].data_structure)
{
  "type": "monotonic",
  "coordinates": [
    "1 s",
    "5 s",
    "10 s",
    "20 s",
    "40 s",
    "80 s"
  ],
  "quantity_name": "time",
  "label": "t1",
  "reciprocal": {
    "quantity_name": "frequency"
  }
}
```

respectively. The first dimension is uniformly spaced, as indicated by the *linear* subtype, while the second dimension is non-linear and monotonically sampled. The coordinates along the respective dimensions are

```
>>> x0 = x[0].coordinates
>>> print(x0)
[-41040. -40960. -40880. ...  40640.  40720.  40800.] us

>>> x1 = x[1].coordinates
>>> print(x1)
[ 1.  5. 10. 20. 40. 80.] s
```

Notice, the unit of `x0` is in microseconds. It might be convenient to convert the unit to milliseconds. To do so, use the *to()* method of the respective *Dimension* instance as follows,

```
>>> x[0].to('ms')
>>> x0 = x[0].coordinates
>>> print(x0)
[-41.04 -40.96 -40.88 ...  40.64  40.72  40.8 ] ms
```

As before, the components of the dependent variable are accessed using the *components* attribute.

```
>>> y00 = y[0].components[0]
>>> print(y00)
[[  182.26953   +136.4989j    -530.45996   +145.59097j
   -648.56055   +296.6433j   ... -1034.6655    +123.473114j
    137.29883   +144.3381j    -151.75049    -18.316727j]
 [  -80.799805  +138.63733j   -330.4419    -131.69786j
   -356.23877   +463.6406j   ...   854.9712    +373.60577j
    432.64648   +525.6024j     -35.51758   -141.60239j ]
 [ -215.80469   +163.03308j   -330.6836    -308.8578j
  -1313.7393   -1557.9144j   ...  -979.9209    +271.06757j
   -667.6211     +61.262817j   150.32227    -41.081024j]
 [    6.2421875 -163.0319j    -654.5654    +372.27518j
  -1209.3877    -217.7103j   ...   202.91211   +910.0657j
   -163.88281   +343.41882j     27.354492   +21.467224j]
 [  -86.03516   -129.40945j   -461.1875     -74.49284j
    68.13672   -641.11975j   ...   803.3242    -423.6355j
   -267.3672    -226.39514j     77.77344    +80.2041j  ]
 [ -436.0664    -131.52814j    216.32812   +441.56696j
   -577.0254    -658.17645j   ... -1780.457     +454.20862j
  -1765.7441    -375.72888j    407.0703    +162.24716j ]]
```

Visualizing the dataset

Tip: Intensity plot with cross-sections

More often than not, the code required to plot the data become exhaustive. Here is one such example.

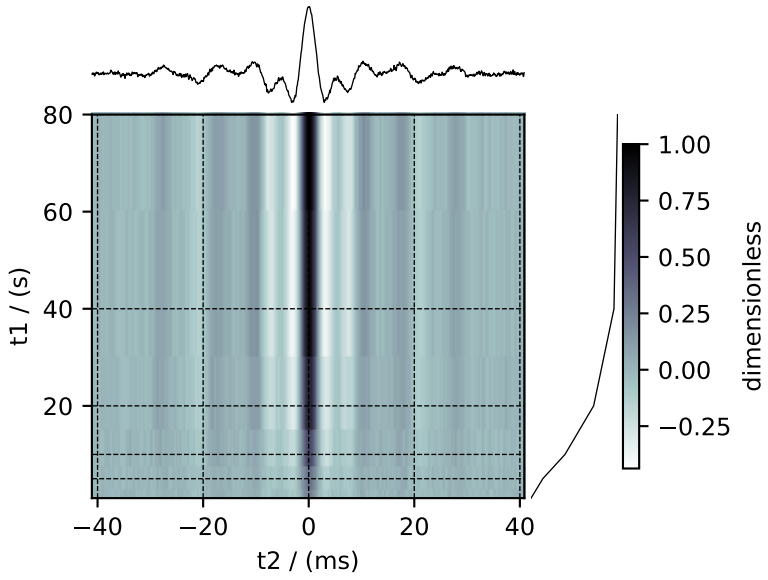
```
>>> import matplotlib.pyplot as plt
>>> from matplotlib.image import NonUniformImage
>>> import numpy as np

>>> def plot_nmr_2d():
...     """
...     Set the extents of the image.
...     To set the independent variable coordinates at the center of each image
...     pixel, subtract and add half the sampling interval from the first
...     and the last coordinate, respectively, of the linearly sampled
...     dimension, i.e., x0.
...     """
...     si=x[0].increment
...     extent = ((x0[0]-0.5*si).to('ms').value,
...               (x0[-1]+0.5*si).to('ms').value,
...               x1[0].value,
...               x1[-1].value)
...
...     """
...     Create a 2x2 subplot grid. The subplot at the lower-left corner is for
...     the image intensity plot. The subplots at the top-left and bottom-right
...     are for the data slice at the horizontal and vertical cross-section,
...     respectively. The subplot at the top-right corner is empty.
```

(continues on next page)

(continued from previous page)

```
...     """
...     fig, axi = plt.subplots(2,2, figsize=(4,3),
...                             gridspec_kw = {'width_ratios':[4,1],
...                                            'height_ratios':[1,4]})
...
...     """
...     The image subplot quadrant.
...     Add an image over a rectilinear grid. Here, only the real part of the
...     data values is used.
...     """
...     ax = axi[1,0]
...     im = NonUniformImage(ax, interpolation='nearest',
...                          extent=extent, cmap='bone_r')
...     im.set_data(x0, x1, y00.real/y00.real.max())
...
...     """Add the colorbar and the component label."""
...     cbar = fig.colorbar(im)
...     cbar.ax.set_ylabel(y[0].axis_label[0])
...
...     """Set up the grid lines."""
...     ax.images.append(im)
...     for i in range(x1.size):
...         ax.plot(x0, np.ones(x0.size)*x1[i], 'k--', linewidth=0.5)
...     ax.grid(axis='x', color='k', linestyle='--', linewidth=0.5, which='both')
...
...     """Setup the axes, add the axes labels, and the figure title."""
...     ax.set_xlim([extent[0], extent[1]])
...     ax.set_ylim([extent[2], extent[3]])
...     ax.set_xlabel(x[0].axis_label)
...     ax.set_ylabel(x[1].axis_label)
...     ax.set_title(y[0].name)
...
...     """Add the horizontal data slice to the top-left subplot."""
...     ax0 = axi[0,0]
...     top = y00[-1].real
...     ax0.plot(x0, top, 'k', linewidth=0.5)
...     ax0.set_xlim([extent[0], extent[1]])
...     ax0.set_ylim([top.min(), top.max()])
...     ax0.axis('off')
...
...     """Add the vertical data slice to the bottom-right subplot."""
...     ax1 = axi[1,1]
...     right = y00[:,513].real
...     ax1.plot(right, x1, 'k', linewidth=0.5)
...     ax1.set_ylim([extent[2], extent[3]])
...     ax1.set_xlim([right.min(),  right.max()])
...     ax1.axis('off')
...
...     """Turn off the axis system for the top-right subplot."""
...     axi[0,1].axis('off')
...
...     plt.tight_layout(pad=0., w_pad=0., h_pad=0.)
...     plt.subplots_adjust(wspace=0.025, hspace=0.05)
...     plt.show()
```

```
>>> plot_nmr_2d()
```



4.2.3 Transmission Electron Microscopy (TEM) dataset

The following TEM dataset is a section of an early larval brain of *Drosophila melanogaster* used in the analysis of neuronal microcircuitry. The dataset was obtained from the TrakEM2 tutorial and subsequently converted to the CSD model file-format.

Let's import the CSD model data-file and look at its data structure.

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

>>> filename = 'Test Files/TEM/TEM.csdf'
>>> TEM = cp.load(filename)
>>> print(TEM.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2016-03-12T16:41:00Z",
    "description": "TEM image of the early larval brain of Drosophila melanogaster␣
↪used in the analysis of neuronal microcircuitry.",
    "dimensions": [
      {
        "type": "linear",
        "count": 512,
        "increment": "4.0 nm",
        "quantity_name": "length",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      },
      {
        "type": "linear",
```

(continues on next page)

(continued from previous page)

```
        "count": 512,
        "increment": "4.0 nm",
        "quantity_name": "length",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "numeric_type": "uint8",
        "quantity_type": "scalar",
        "components": [
          [
            "126, 107, ..., 164, 171"
          ]
        ]
      }
    ]
  }
}
```

This dataset consists of two linear dimensions and one single-component dependent variable. The tuples of the dimension and the dependent variable instances from this example are

```
>>> x = TEM.dimensions
>>> y = TEM.dependent_variables
```

and the respective coordinates (viewed only for the first ten coordinates),

```
>>> print(x[0].coordinates[:10])
[ 0.  4.  8. 12. 16. 20. 24. 28. 32. 36.] nm

>>> print(x[1].coordinates[:10])
[ 0.  4.  8. 12. 16. 20. 24. 28. 32. 36.] nm
```

For convenience, let's convert the coordinate from *nm* to *µm* using the `to()` method of the respective *Dimension* instance,

```
>>> x[0].to('µm')
>>> x[1].to('µm')
```

and plot the data.

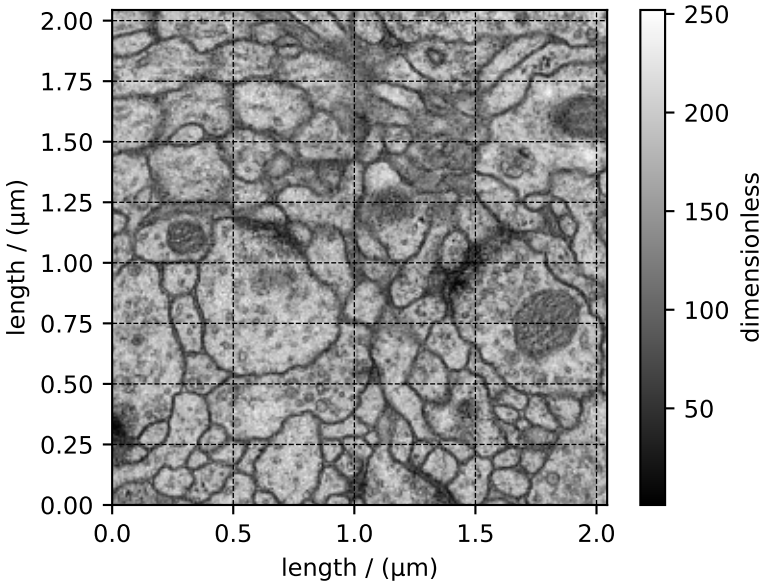
```
>>> def plot_image():
...     plt.figure(figsize=(4,3))
...
...     # Set the extents of the image plot.
...     extent = [x[0].coordinates[0].value, x[0].coordinates[-1].value,
...               x[1].coordinates[0].value, x[1].coordinates[-1].value]
...
...     # Add the image plot.
...     im = plt.imshow(y[0].components[0], origin='lower', extent=extent, cmap='gray
↪')
...
...     # Add a colorbar.
```

(continues on next page)

(continued from previous page)

```
...     cbar = plt.gca().figure.colorbar(im)
...     cbar.ax.set_ylabel(y[0].axis_label[0])
...
...     # Set up the axes label and figure title.
...     plt.xlabel(x[0].axis_label)
...     plt.ylabel(x[1].axis_label)
...     plt.title(y[0].name)
...
...     # Set up the grid lines.
...     plt.grid(color='k', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot_image()
```



4.3 Vector, 2D{2} datasets

The 2D{2} datasets are two-dimensional, $d = 2$, with one two-component dependent variable, $p = 2$.

4.3.1 Vector dataset

The following is an example of a simulated electric field vector dataset of a dipole as a function of two linearly sampled spatial dimensions.

```
>>> import csdmpy as cp

>>> filename = 'Test Files/vector/electric_field/electric_field_raw.csdfe'
>>> vector_data = cp.load(filename)
>>> print (vector_data.data_structure)
{
```

(continues on next page)

(continued from previous page)


```
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2014-09-30T11:16:33Z",
    "description": "A simulated electric field dataset from an electric dipole.",
    "dimensions": [
      {
        "type": "linear",
        "count": 64,
        "increment": "0.0625 cm",
        "coordinates_offset": "-2.0 cm",
        "quantity_name": "length",
        "label": "x",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      },
      {
        "type": "linear",
        "count": 64,
        "increment": "0.0625 cm",
        "coordinates_offset": "-2.0 cm",
        "quantity_name": "length",
        "label": "y",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "Electric field lines",
        "unit": "C^-1 * N",
        "quantity_name": "electric field strength",
        "numeric_type": "float32",
        "quantity_type": "vector_2",
        "components": [
          [
            "3.7466873e-07, 3.3365018e-07, ..., 3.5343004e-07, 4.0100363e-07"
          ],
          [
            "1.6129676e-06, 1.6765767e-06, ..., 1.846712e-06, 1.7754871e-06"
          ]
        ]
      }
    ]
  }
}
```


The tuples of the dimension and dependent variable instances from this example are

```
>>> x = vector_data.dimensions
>>> y = vector_data.dependent_variables
```

with the respective coordinates (viewed only up to five values), as

```
>>> print(x[0].coordinates[:5])
[-2.     -1.9375 -1.875  -1.8125 -1.75  ] cm

>>> print(x[1].coordinates[:5])
[-2.     -1.9375 -1.875  -1.8125 -1.75  ] cm
```

In this example, the components of the dependent variable are vectors as seen from the *quantity_type* attribute of the corresponding dependent variable instance.

```
>>> print(y[0].quantity_type)
vector_2
```

From the value *vector_2*, *vector* indicates a vector dataset, while *2* indicates the number of vector components.

Visualizing the dataset

Let's visualize the vector data using the *streamplot* method from the matplotlib package. Before we could visualize, however, there is an initial processing step. We use the Numpy library for processing.

```
>>> import numpy as np

>>> X, Y = np.meshgrid(x[0].coordinates, x[1].coordinates)
>>> U, V = y[0].components[0], y[0].components[1]
>>> R = np.sqrt(U**2 + V**2)
>>> R/=R.min()
>>> Rlog=np.log10(R)
```

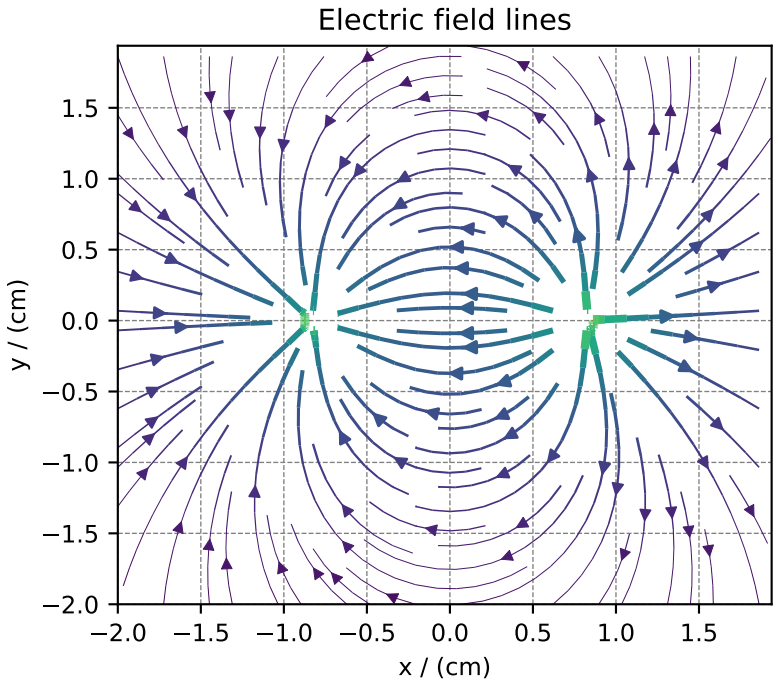
In the above steps, we calculate the X-Y grid points along with a scaled magnitude of the vector dataset. The magnitude is scaled such that the minimum value is one. Next, calculate the log of the scaled magnitude to visualize the intensity on a logarithmic scale.

And now, the plot.

Tip: Streamplot vector visualization

```
>>> import matplotlib.pyplot as plt
>>> def plot_vector():
...     plt.figure(figsize=(4,3.5))
...     plt.streamplot(X.value, Y.value, U, V, density =1,
...                    linewidth=Rlog, color=Rlog, cmap='viridis')
...
...     plt.xlim([x[0].coordinates[0].value, x[0].coordinates[-1].value])
...     plt.ylim([x[1].coordinates[0].value, x[1].coordinates[-1].value])
...
...     # Set axes labels and figure title.
...     plt.xlabel(x[0].axis_label)
...     plt.ylabel(x[1].axis_label)
...     plt.title(y[0].name)
...
...     # Set grid lines.
...     plt.grid(color='gray', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot_vector()
```



4.4 Pixel, 2D{3} datasets

The 2D{3} datasets is two dimensional, $d = 2$, with a single three-component dependent variable, $p = 3$.

4.4.1 Image datasets

A common example from this subset is perhaps the RGB image dataset. An RGB image dataset has two spatial dimensions and one dependent variable with three components corresponding to the red, green, and blue color intensities.

The following is an example of the RGB image dataset.

```
>>> import csdmpy as cp

>>> filename = 'Test Files/image/raccoon_image.csdf'
>>> ImageData = cp.load(filename)
>>> print (ImageData.data_structure)
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2016-03-12T16:41:00Z",
    "tags": [
      "racoon",
      "image",
      "Judy Weggelaar"
    ],
    "description": "An RBG image of a raccoon face.",
```

(continues on next page)

(continued from previous page)

```
    "dimensions": [
      {
        "type": "linear",
        "count": 1024,
        "increment": "1.0",
        "label": "horizontal index"
      },
      {
        "type": "linear",
        "count": 768,
        "increment": "1.0",
        "label": "vertical index"
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "name": "raccoon",
        "numeric_type": "uint8",
        "quantity_type": "pixel_3",
        "component_labels": [
          "red",
          "green",
          "blue"
        ],
        "components": [
          [
            "121, 138, ..., 119, 118"
          ],
          [
            "112, 129, ..., 155, 154"
          ],
          [
            "131, 148, ..., 93, 92"
          ]
        ]
      }
    ]
  }
}
```

The tuples of the dimension and dependent variable instances from `ImageData` instance are

```
>>> x = ImageData.dimensions
>>> y = ImageData.dependent_variables
```

respectively. There are two dimensions, and the coordinates along each dimension are

```
>>> print('x0 =', x[0].coordinates[:10])
x0 = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]

>>> print('x1 =', x[1].coordinates[:10])
x1 = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

respectively. In the above example, only the first ten coordinates along each dimension are displayed.

The dependent variable is the image data, as also seen from the *quantity_type* attribute of the corresponding *DependentVariable* instance.

```
>>> print(y[0].quantity_type)
pixel_3
```

From the value *pixel_3*, *pixel* indicates a pixel data point, while *3* indicates the number of pixel components.

As usual, the components of the dependent variable are accessed through the *components* attribute. To access the individual components, use the appropriate array indexing. For example,

```
>>> print (y[0].components[0])
[[121 138 153 ... 119 131 139]
 [ 89 110 130 ... 118 134 146]
 [ 73  94 115 ... 117 133 144]
 ...
 [ 87  94 107 ... 120 119 119]
 [ 85  95 112 ... 121 120 120]
 [ 85  97 111 ... 120 119 118]]
```

will return an array with the first component of all data values. In this case, the components correspond to the red color intensity, also indicated by the corresponding component label. The label corresponding to this component array is accessed through the *component_labels* attribute with appropriate indexing, that is

```
>>> print (y[0].component_labels[0])
red
```

To avoid displaying larger output, as an example, we print the shape of each component array (using Numpy array's *shape* attribute) for the three components along with their respective labels.

```
>>> print (y[0].component_labels[0], y[0].components[0].shape)
red (768, 1024)

>>> print (y[0].component_labels[1], y[0].components[1].shape)
green (768, 1024)

>>> print (y[0].component_labels[2], y[0].components[2].shape)
blue (768, 1024)
```

The shape (768, 1024) corresponds to the number of points from the each dimension instances.

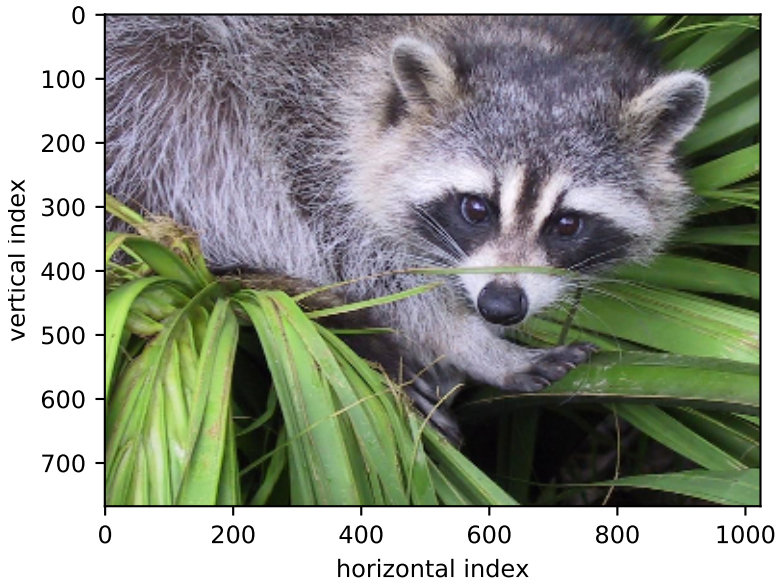
Note: In this example, since there is only one dependent variable, the index of *y* is set to zero, which is `y[0]`. The indices for the *components* and the *component_labels*, on the other hand, spans through the number of components.

Now, to visualize the dataset as an RGB image, we use the matplotlib *imshow* method.

Tip: RGB image plot

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

>>> def image_data():
...     fig, ax = plt.subplots(1,1, figsize=(4,3))
...     ax.imshow(np.moveaxis(y[0].components, 0, -1 ))
...     ax.set_xlabel(x[0].axis_label)
...     ax.set_ylabel(x[1].axis_label)
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> image_data()
```



4.5 Correlated Dataset

The Core Scientific Dataset Model (CSDM) supports multiple dependent variables that share the same d-dimensional coordinate grid, where $d >= 0$. We call the dependent variables from these datasets as *correlated datasets*.

In this section, we go over a few examples.

4.5.1 Scatter, 0D{1,1} dataset

We start with a 0D{1,1} correlated dataset, that is, a dataset without a coordinate grid. A 0D{1,1} dataset has no dimensions, d = 0, and two single-component dependent variables. In the following example, the two *correlated* dependent variables are the ^{29}Si - ^{29}Si nuclear spin couplings, ^{2}J, across a Si-O-Si linkage, and the s-character product on the O and two Si along the Si-O bond across the Si-O-Si linkage.

Let's import the dataset.

```
>>> import csdmpy as cp
>>> filename = 'Test Files/correlatedDataset/0D_dataset/J_vs_s.csdf'
>>> zero_d_dataset = cp.load(filename)
```

Since the dataset has no dimensions, the value of the *dimensions* attribute of the *CSDM* class is an empty tuple,

```
>>> print(zero_d_dataset.dimensions)
()
```

The *dependent_variables* attribute, however, holds two dependent-variable objects. The data structure from the two dependent variables is

```
>>> print(zero_d_dataset.dependent_variables[0].data_structure)
{
  "type": "internal",
  "name": "Gaussian computed J-couplings",
  "unit": "Hz",
  "quantity_name": "frequency",
  "numeric_type": "float32",
  "quantity_type": "scalar",
  "component_labels": [
    "J-coupling"
  ],
  "components": [
    [
      "-1.87378, -1.42918, ..., 25.1742, 26.0608"
    ]
  ]
}
```

and

```
>>> print(zero_d_dataset.dependent_variables[1].data_structure)
{
  "type": "internal",
  "name": "product of s-characters",
  "unit": "%",
  "numeric_type": "float32",
  "quantity_type": "scalar",
  "component_labels": [
    "s-character product"
  ],
  "components": [
    [
      "0.8457453, 0.8534185, ..., 1.5277092, 1.5289451"
    ]
  ]
}
```

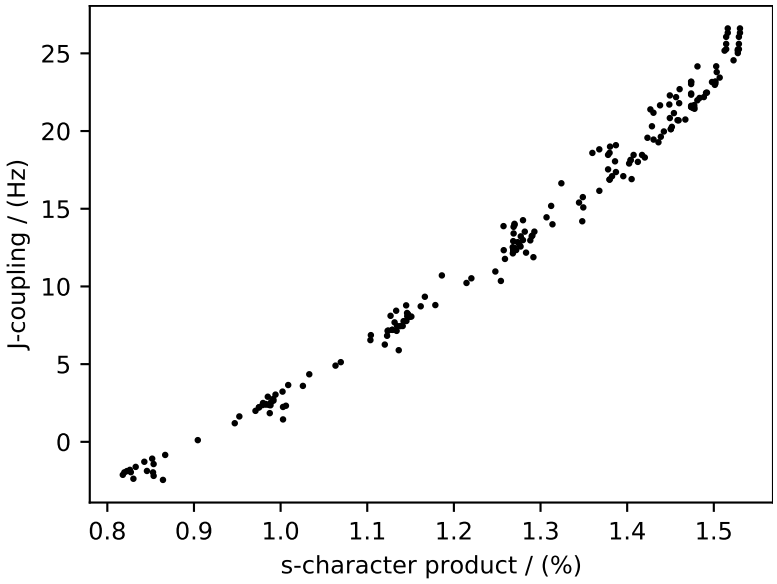
respectively.

The correlation plot of the dependent-variables from the dataset is shown below.

Tip: Plotting a scatter plot.

```
>>> import matplotlib.pyplot as plt
>>> def plot_scatter():
...     plt.figure(figsize=(4,3))
...
...     y0 = zero_d_dataset.dependent_variables[0]
...     y1 = zero_d_dataset.dependent_variables[1]
...
...     plt.scatter(y1.components[0], y0.components[0], s=2, c='k')
...     plt.xlabel(y1.axis_label[0])
...     plt.ylabel(y0.axis_label[0])
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot_scatter()
```



4.5.2 Meteorological, 2D{1,1,2,1,1} dataset

The following dataset is obtained from NOAA/NCEP Global Forecast System (GFS) Atmospheric Model and subsequently converted to the CSD model file-format. The dataset consists of two spatial dimensions describing the geographical coordinates of the earth surface and five dependent variables with 1) surface temperature, 2) air temperature at 2 m, 3) relative humidity, 4) air pressure at sea level as the four *scalar* quantity_type dependent variable, and 5) wind velocity as the two-component *vector*, quantity_type dependent variable.

Let's import the *csdmpy* module and load this dataset.

```
>>> import csdmpy as cp

>>> filename = 'Test Files/correlatedDataset/forecast/NCEI.csdfe'
>>> multi_dataset = cp.load(filename)
```

The tuple of dimension and dependent variable objects from `multi_dataset` instance are

```
>>> x = multi_dataset.dimensions
>>> y = multi_dataset.dependent_variables
```

The dataset contains two dimension objects representing the *longitude* and *latitude* of the earth's surface. The respective dimensions are labeled as

```
>>> x[0].label
'longitude'

>>> x[1].label
'latitude'
```

There are a total of five dependent variables stored in this dataset. The first dependent variable is the surface air temperature. The data structure of this dependent variable is

```
>>> print(y[0].data_structure)
{
  "type": "internal",
  "description": "The label 'tmpsfc' is the standard attribute name for 'surface air␣
↪temperature'.",
  "name": "Surface temperature",
  "unit": "K",
  "quantity_name": "temperature",
  "numeric_type": "float64",
  "quantity_type": "scalar",
  "component_labels": [
    "tmpsfc - surface air temperature"
  ],
  "components": [
    [
      "292.8152160644531, 293.0152282714844, ..., 301.8152160644531, 303.8152160644531
↪"
    ]
  ]
}
```

If you have followed all previous examples, the above data structure should be self-explanatory.

The following snippet plots a dependent variable of scalar *quantity_type*.

Tip: Plotting a scalar intensity plot

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.axes_grid1 import make_axes_locatable

>>> def plot_scalar(yx):
...     fig, ax = plt.subplots(1,1, figsize=(6,3))
...
...     # Set the extents of the image plot.
...     extent = [x[0].coordinates[0].value, x[0].coordinates[-1].value,
...               x[1].coordinates[0].value, x[1].coordinates[-1].value]
...
...     # Add the image plot.
...     im = ax.imshow(yx.components[0], origin='lower', extent=extent,
...                    cmap='coolwarm')
...
...     # Add a colorbar.
...     divider = make_axes_locatable(ax)
...     cax = divider.append_axes("right", size="5%", pad=0.05)
...     cbar = fig.colorbar(im, cax)
...     cbar.ax.set_ylabel(yx.axis_label[0])
...
...     # Set up the axes label and figure title.
...     ax.set_xlabel(x[0].axis_label)
...     ax.set_ylabel(x[1].axis_label)
...     ax.set_title(yx.name)
...
...     # Set up the grid lines.
...     ax.grid(color='k', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
```

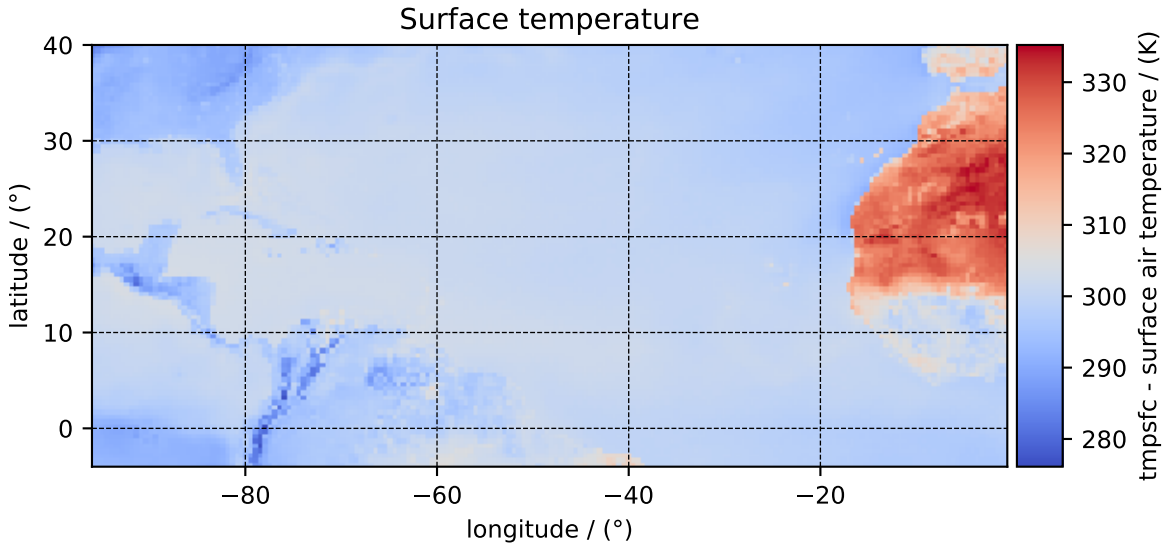
(continues on next page)

(continued from previous page)

```
...     plt.show()
```

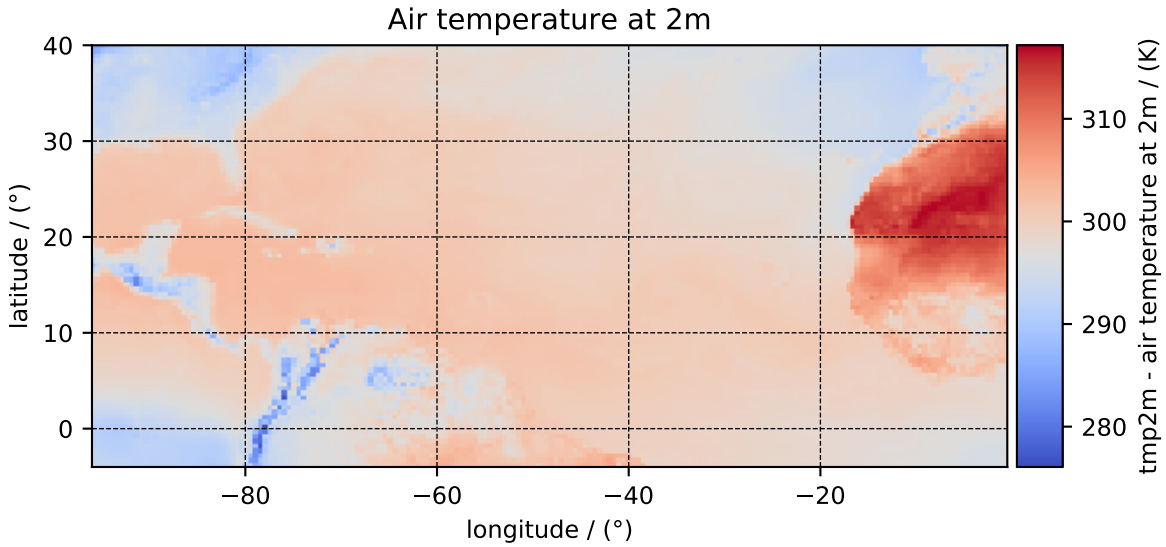
Now to plot the data from the dependent variable.

```
>>> plot_scalar(y[0])
```

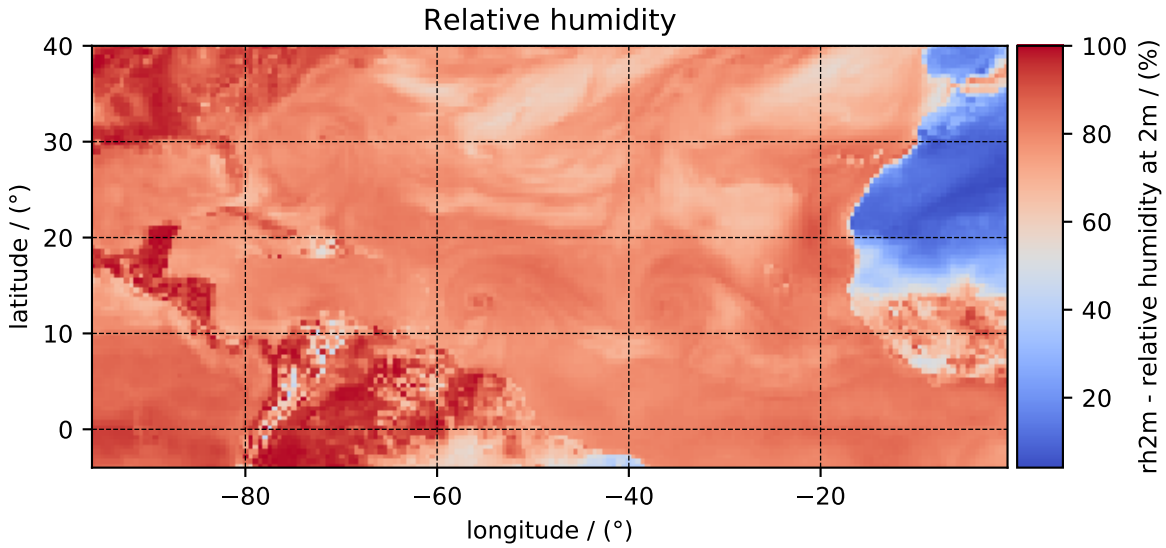


Similarly, other dependent variables with their respective plots are

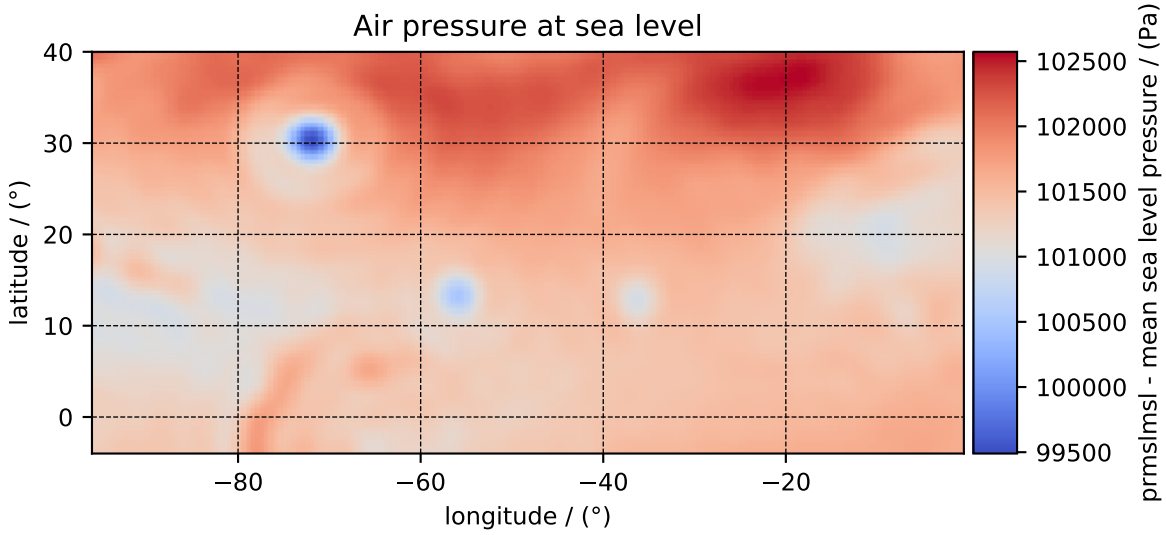
```
>>> y[1].name
'Air temperature at 2m'
>>> plot_scalar(y[1])
```



```
>>> y[3].name
'Relative humidity'
>>> plot_scalar(y[3])
```




```
>>> y[4].name
'Air pressure at sea level'
>>> plot_scalar(y[4])
```



Notice, we skipped the dependent variable at index two. The reason is that this particular dependent variable is a vector dataset,

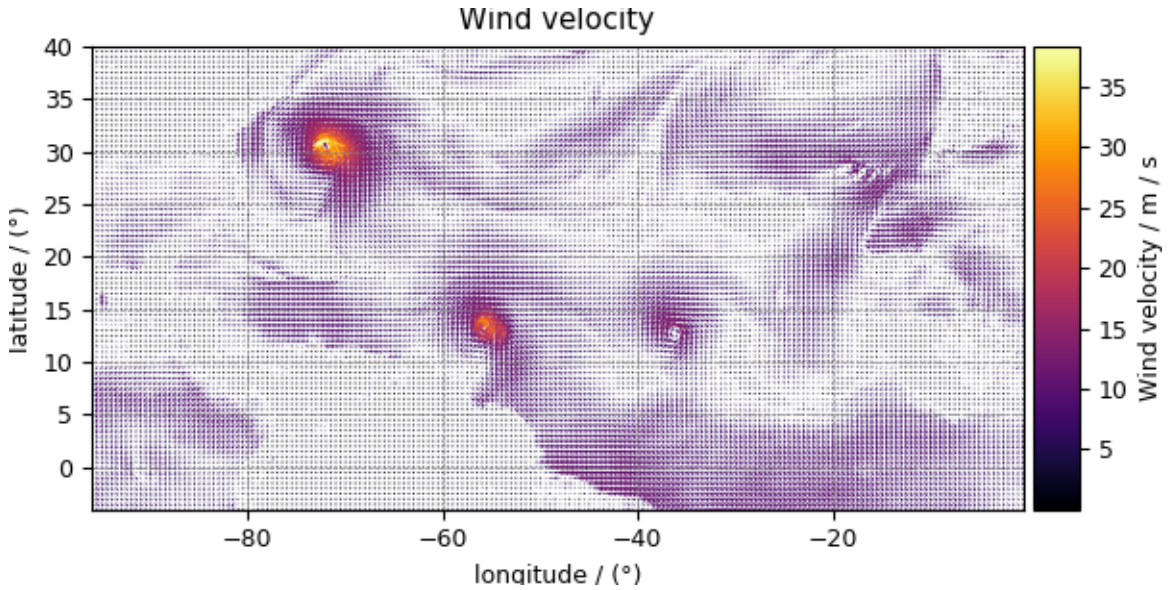
```
>>> y[2].quantity_type
'vector_2'
>>> y[2].name
'Wind velocity'
```

which represents the wind velocity, and requires a vector visualization routine. To visualize the vector data, we use the matplotlib quiver plot.

Tip: Vector quiver plot

```
>>> def plot_vector(yx):
...     fig, ax = plt.subplots(1,1, figsize=(6,3))
...     X, Y = np.meshgrid(x[0].coordinates, x[1].coordinates)
...     magnitude = np.sqrt(yx.components[0]**2 + yx.components[1]**2)
...
...     cf = ax.quiver(x[0].coordinates, x[1].coordinates,
...                    yx.components[0], yx.components[1],
...                    magnitude, pivot ='middle', cmap='inferno')
...     divider = make_axes_locatable(ax)
...     cax = divider.append_axes("right", size="5%", pad=0.05)
...     cbar = fig.colorbar(cf, cax)
...     cbar.ax.set_ylabel(yx.name+' / '+str(yx.unit))
...
...     ax.set_xlim([x[0].coordinates[0].value, x[0].coordinates[-1].value])
...     ax.set_ylim([x[1].coordinates[0].value, x[1].coordinates[-1].value])
...
...     # Set axes labels and figure title.
...     ax.set_xlabel(x[0].axis_label)
...     ax.set_ylabel(x[1].axis_label)
...     ax.set_title(yx.name)
...
...     # Set grid lines.
...     ax.grid(color='gray', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot_vector(y[2])
```



4.5.3 Astronomy, 2D{1,1,1} dataset (Creating image composition)

More often, the images in astronomy are a composition of datasets measured at different wavelengths over an area of the sky. In this example, we illustrate the use of the CSDM file-format, and *csdmpy* module, beyond just reading a CSDM-compliant file. We'll use these datasets, and compose an image, using Numpy arrays. The following example is the data from the *Eagle Nebula* acquired at three different wavelengths and serialized as a CSDM compliant file. Import the *csdmpy* model and load the dataset.

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

>>> filename = 'Test Files/EagleNebula/eagleNebula.csdfe'
>>> eagle_nebula = cp.load(filename)
```

Let's get the tuple of dimension and dependent variable objects from the `eagle_nebula` instance.

```
>>> x = eagle_nebula.dimensions
>>> y = eagle_nebula.dependent_variables
```

Before we compose an image, let's take a look at the individual dependent variables from the dataset. The three dependent variables correspond to signal acquisition at 502 nm, 656 nm, and 673 nm, respectively. This information is also listed in the *name* attribute of the respective dependent variable instances,

```
>>> y[0].name
'Eagle Nebula acquired @ 502 nm'
>>> y[1].name
'Eagle Nebula acquired @ 656 nm'
>>> y[2].name
'Eagle Nebula acquired @ 673 nm'
```

Tip: A script for an intensity plot.

```
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.axes_grid1 import make_axes_locatable
>>> from matplotlib.colors import LogNorm

>>> def plot_scalar(yx):
...     plt.figure(figsize=(6,4.5))
...
...     # Set the extents of the image plot.
...     extent = [x[0].coordinates[0].value, x[0].coordinates[-1].value,
...               x[1].coordinates[0].value, x[1].coordinates[-1].value]
...
...     # Add the image plot.
...     y0 = yx.components[0]
...     y0 = y0/y0.max()
...     im = plt.imshow(y0, origin='lower', extent=extent, cmap='bone', vmax=0.1)
...
...     # Add a colorbar.
...     divider = make_axes_locatable(plt.gca())
...     cax = divider.append_axes("right", size="5%", pad=0.05)
...     cbar = plt.gca().figure.colorbar(im, cax)
...     cbar.ax.set_ylabel(yx.axis_label[0])
...
...     # Set up the axes label and figure title.
...     plt.xlabel(x[0].axis_label)
```

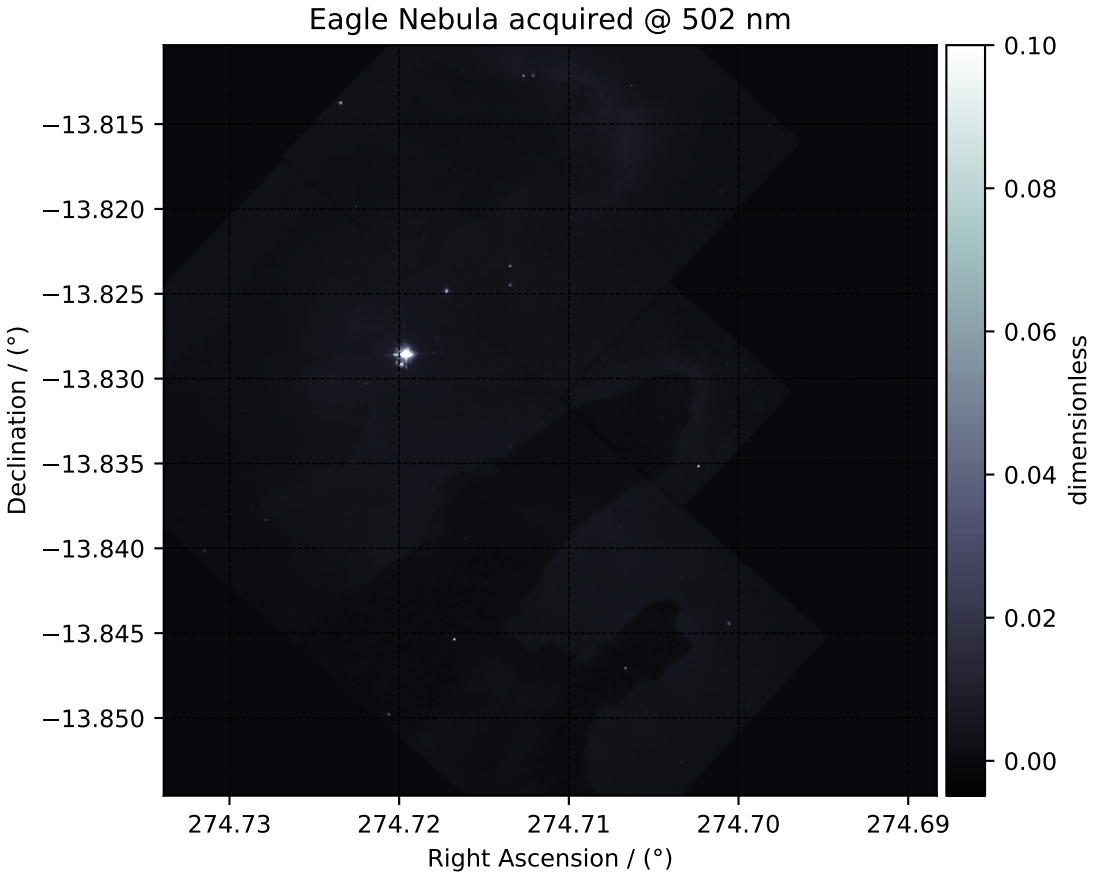
(continues on next page)

(continued from previous page)

```
...     plt.ylabel(x[1].axis_label)
...     plt.title(yx.name)
...
...     # Set up the grid lines.
...     plt.grid(color='k', linestyle='--', linewidth=0.5)
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

Let's plot the dependent variables, first dependent variable,

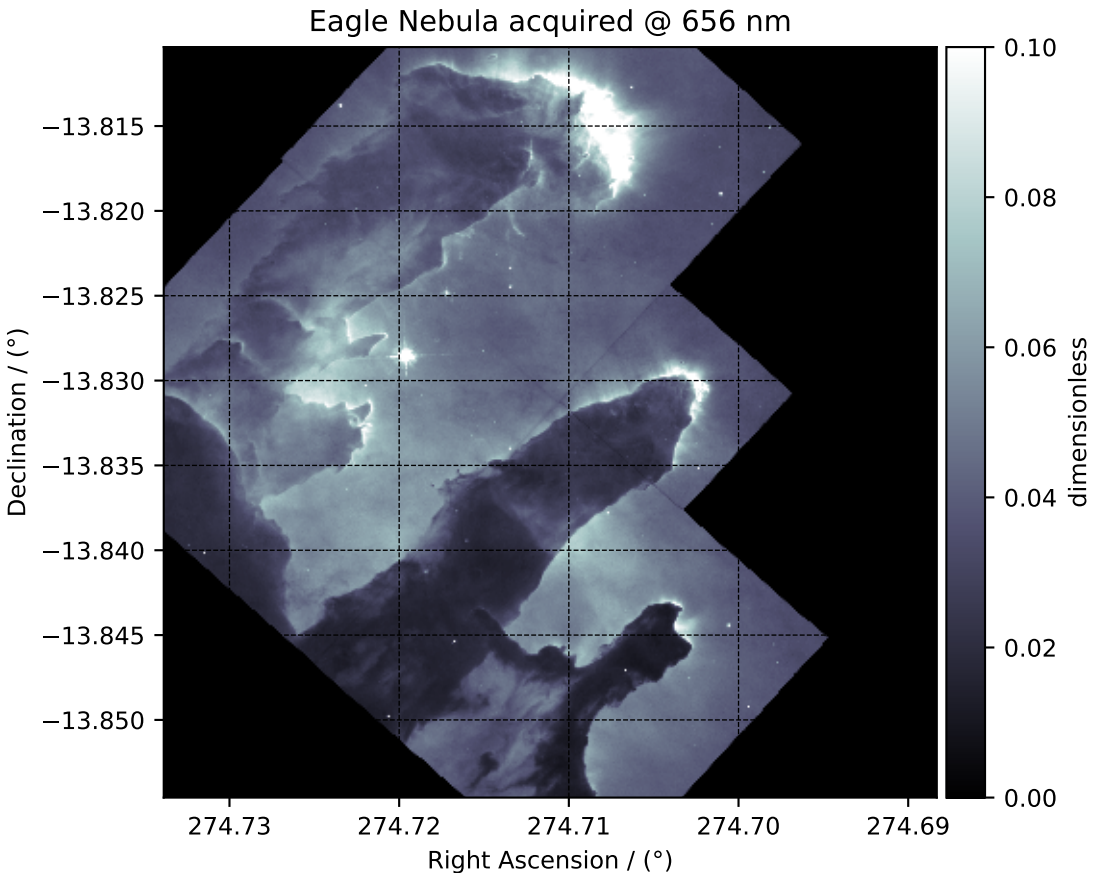
```
>>> plot_scalar(y[0])
```



second dependent variable, and

```
>>> plot_scalar(y[1])
```

the third dependent variable.




```
>>> plot_scalar(y[2])
```

Image composition

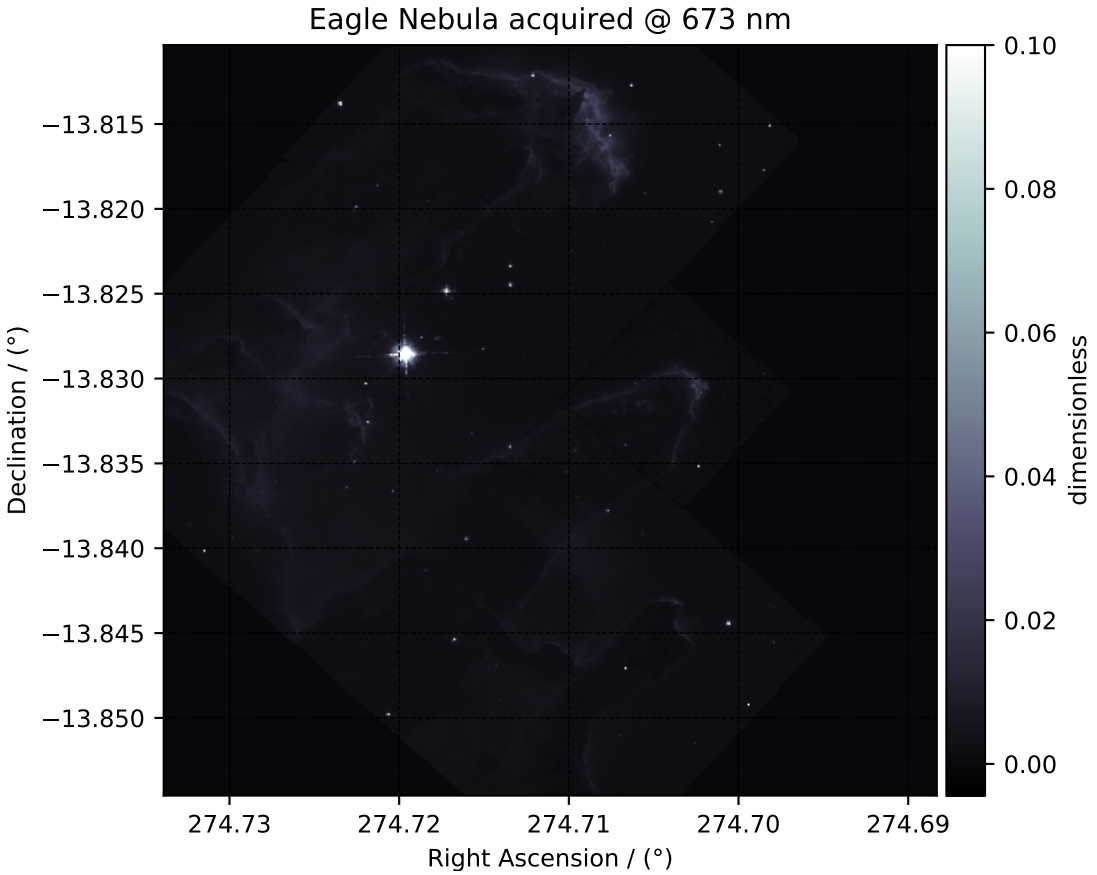
For the image composition, we assign the dependent variable at index zero as the blue channel, index one as the green channel, and index two as the red channel of an RGB image. Start with creating an empty array to hold the RGB dataset.

```
>>> shape = y[0].components[0].shape + (3,)
>>> image = np.empty(shape, dtype=np.float64)
```

Here, `image` is the variable we use for storing the composition. Add the respective dependent variables to the designated color channel in the `image` array,

```
>>> image[...,0] = y[2].components[0]/y[2].components[0].max() # red channel
>>> image[...,1] = y[1].components[0]/y[1].components[0].max() # green channel
>>> image[...,2] = y[0].components[0]/y[0].components[0].max() # blue channel
```

Following the intensity plot of the individual dependent variables, see the above figures, it is evident that the component intensity from `y[1]` and, therefore, the green channel dominates the other two. If we plot the `image` data, the image will be saturated with green intensity. To attain a color-balanced image, we arbitrarily scale the intensities from the three channels. You may choose any scaling factor. Each scaling factor will produce a different composition. In this example, we use the following,

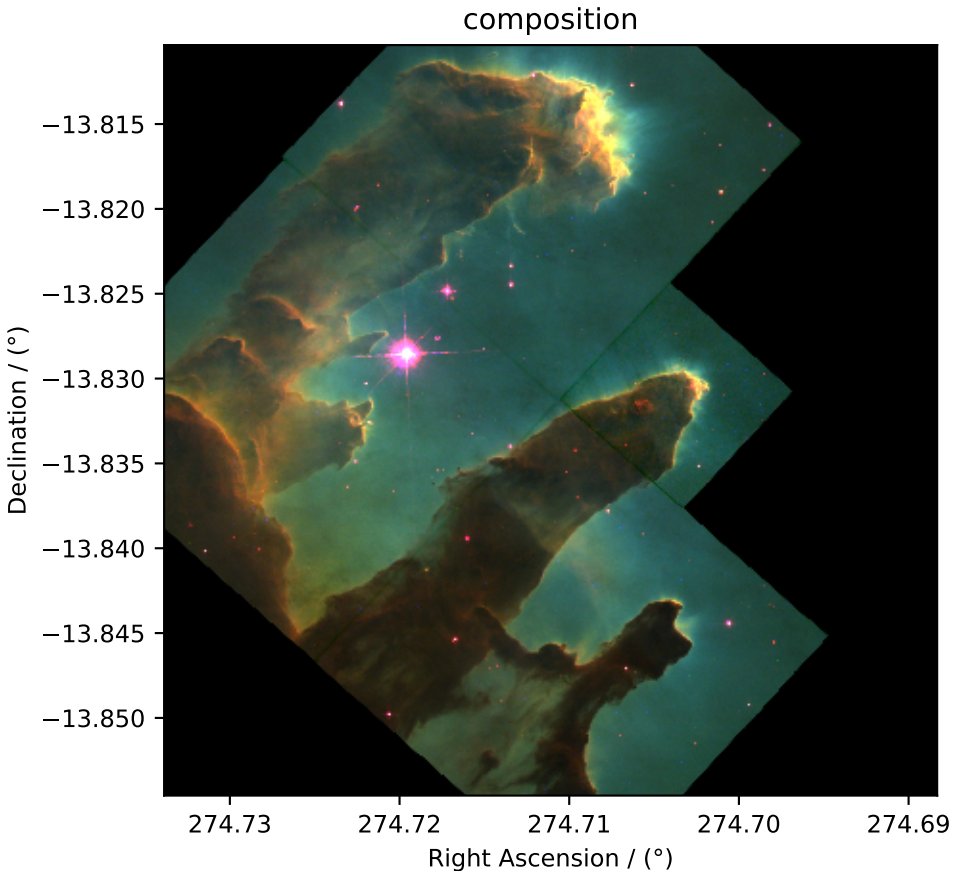



```
>>> image[...,0] = image[...,0]*65.0 # red channel
>>> image[...,1] = image[...,1]*7.5  # green channel
>>> image[...,2] = image[...,2]*75.0 # blue channel
```

Now to plot this composition.

```
>>> def image_composition():
...     # Set the extents of the image plot.
...     extent = [x[0].coordinates[0].value, x[0].coordinates[-1].value,
...               x[1].coordinates[0].value, x[1].coordinates[-1].value]
...
...     # add figure
...     plt.figure(figsize=(5,4.5))
...     plt.imshow(image, origin='lower', extent=extent)
...
...     plt.xlabel(x[0].axis_label)
...     plt.ylabel(x[1].axis_label)
...     plt.title('composition')
...
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> image_composition()
```



4.6 Labeled Dataset

The CSD model also supports labeled dimensions. In the following example, we present a mixed *linear* and *labeled* two-dimensional dataset representing the population of the country as a function of year. The dataset is obtained from The World Bank.

Import the *csdmpy* model and load the dataset.

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

>>> filename = 'Test Files/labeled/population.csdf'
>>> labeled_data = cp.load(filename)
```

The tuple of dimension and dependent variable objects from `labeled_data` instance are

```
>>> x = labeled_data.dimensions
>>> y = labeled_data.dependent_variables
```

Since one of the dimensions is a *labeled* dimension, let's make use of the *type* attribute of the dimension instances to find out which dimension is *labeled*.

```
>>> x[0].type
'linear'
>>> x[1].type
'labeled'
```

Look like the second dimension is a *labeled* dimension with[1]

```
>>> x[1].count
263
```

labels, where the first five labels are

```
>>> print(x[1].labels[:5])
['Aruba' 'Afghanistan' 'Angola' 'Albania' 'Andorra']
```

Note: For labeled dimensions, the *coordinates* attribute is an alias of the *labels* attribute. Therefore,

```
>>> print(x[1].coordinates[:5])
['Aruba' 'Afghanistan' 'Angola' 'Albania' 'Andorra']
```

The coordinates along the first dimension viewed up to the first ten points are

```
>>> print(x[0].coordinates[:10])
[1960. 1961. 1962. 1963. 1964. 1965. 1966. 1967. 1968. 1969.] yr
```

Plotting the dataset

You may plot this dataset however you like. Here, we use a bar graph to represent the population of countries in the year 2017. The data corresponding to this year is a cross-section of the dependent variable at index 57 along the `x[0]` dimension.

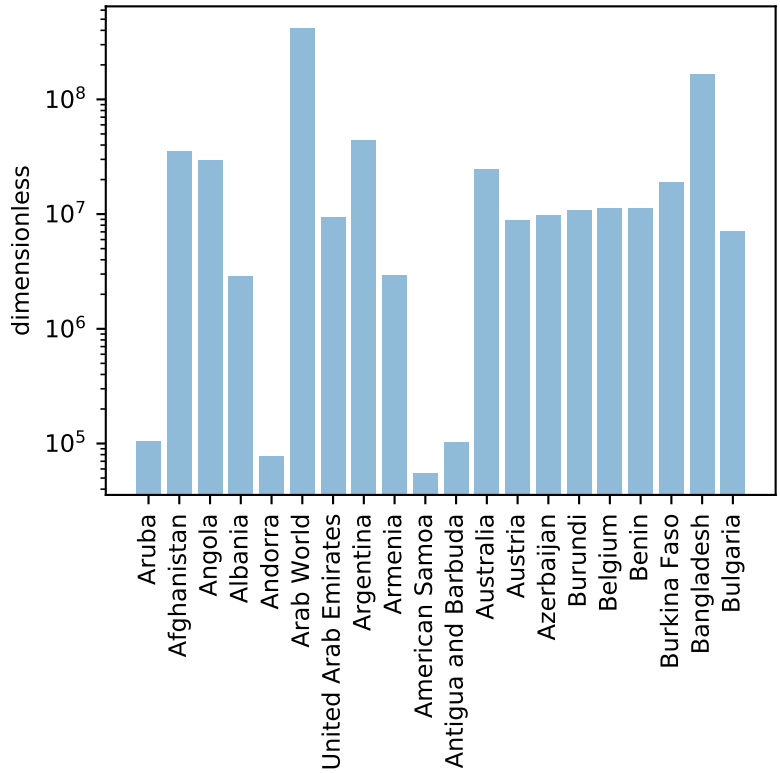
```
>>> print(x[0].coordinates[57])
2017.0 yr
```

To keep the plot simple, we only plot the first 20 country labels along the `x[1]` dimension.

```
>>> def plot_bar():
...     plt.figure(figsize=(4,4))
...
...     x_data = x[1].coordinates[:20]
...     x_pos = np.arange(20)
...     y_data = y[0].components[0][:20, 57]
...
...     plt.bar(x_data, y_data, align='center', alpha=0.5)
...     plt.xticks(x_pos, x_data, rotation=90)
...     plt.ylabel(y[0].axis_label[0])
...     plt.yscale("log")
...     plt.title(y[0].name)
...     plt.tight_layout(pad=0, w_pad=0, h_pad=0)
...     plt.show()
```

```
>>> plot_bar()
```

[1] In the CSD model, the attribute count is only valid for the linearDimension_uml. In csdmpy, however, the *count* attribute is valid for all dimension objects and returns an integer with the number of grid points along the dimension.


dimensionless
10^8
10^7
10^6
10^5
Aruba
Afghanistan
Angola
Albania
Andorra
Arab World
United Arab Emirates
Argentina
Armenia
American Samoa
Antigua and Barbuda
Australia
Austria
Azerbaijan
Burundi
Belgium
Benin
Burkina Faso
Bangladesh
Bulgaria

CHAPTER

FIVE

USING CSDMPY'S OBJECTS

The *csdmpy* module is not just designed for deserializing and serializing the *.csdf* or *.csdfe* files. It can also be used to create new datasets, a feature that is most useful when converting datasets to CSDM compliant files.

5.1 How to create a new dataset

To create a new dataset, import the *csdmpy* module and create a new instance of the CSDM class following,

```
>>> import csdmpy as cp
>>> new_data = cp.new(description='A new test dataset')
```

The *new()* method returns an instance of the CSDM class with zero dimensions and dependent variables. respectively, *i.e.*, a 0D{0} dataset. In the above example, this instance is assigned to the `new_data` variable. Optionally, a description may also be provided as an argument of the *new()* method. The data structure from the above example is

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dataset",
    "dimensions": [],
    "dependent_variables": []
  }
}
```

5.2 How to add instances of Dimension class

In the previous section, we created a new dataset using

```
>>> import csdmpy as cp
>>> new_data = cp.new(description='A new test dimension dataset')
```

In this section, we illustrate, adding dimension objects to a csdm instance. An instance of the Dimension class is added using the *add_dimension()* method of the *CSDM* instance. There are three subtypes of Dimension objects,

- LinearDimension
- MonotonicDimension
- LabeledDimension

See also:

Dimension API for further detail.

5.2.1 LinearDimension

A linear dimension is where the coordinates along the dimension are uniformly spaced. Let's add a LinearDimension instance to the `new_data` instance, using Python's dictionary object as follows

```
>>> d0 = {
...     'type': 'linear',
...     'description': 'This is a linear dimension',
...     'count': 10,
...     'increment': '0.1 s'
... }
```

Here, we define the dimension type as *linear* and provide an *increment*, along with the total number of points, *count*, along the dimension. Now, add this dictionary to the `new_data` instance using

```
>>> new_data.add_dimension(d0)
```

This will generate and add a LinearDimension object to the list of dimensions. The dataset is now a 1D{0} dataset with the following data structure,

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dimension dataset",
    "dimensions": [
      {
        "type": "linear",
        "description": "This is a linear dimension",
        "count": 10,
        "increment": "0.1 s",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
      }
    ],
    "dependent_variables": []
  }
}
```

5.2.2 MonotonicDimension

Try adding another *Dimension* object to this dataset. This time add a monotonic dimension. A monotonic dimension is where the coordinates along the dimension, are spaced either strictly increasing or strictly decreasing. In the following example, we use a different approach for adding the dimension object, that is, using the keyword arguments as follows,

```
>>> new_data.add_dimension(
...     type='monotonic',
...     description='This is a monotonic dimension',
...     coordinates=['1 µG', '2.1 mG', '12.4 G', '0.5 T', '2 T'])
```

The above operation generates an instance of the MonotonicDimension and adds it to the `new_data` instance, thereby, creating a 2D{0} dataset. The data structure form the updated `new_data` instance follows,

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dimension dataset",
    "dimensions": [
      {
        "type": "linear",
        "description": "This is a linear dimension",
        "count": 10,
        "increment": "0.1 s",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
      },
      {
        "type": "monotonic",
        "description": "This is a monotonic dimension",
        "coordinates": [
          "1 µG",
          "2.1 mG",
          "12.4 G",
          "0.5 T",
          "2 T"
        ],
        "quantity_name": "magnetic flux density"
      }
    ],
    "dependent_variables": []
  }
}
```

Notice, every time a new physical dimension is added, the value of the *quantity_name* attribute is appropriately added, if applicable.

5.2.3 LabeledDimension

The third type of dimension is the labeled dimension. As the name suggests, this dimension consists of labels. This type of dimension is useful for datasets describing, for example, the ionization energy as a function of atomic symbols or the population of different countries.

Let's add a labeled dimension to the `new_data` instance. This time pass an instance of the *Dimension* class as the argument of the *add_dimension()* method. To create an instance of the Dimension class follow,

```
>>> from csdmpy import Dimension
>>> d1 = Dimension(
...     type = 'labeled',
...     description = 'This is a labeled dimensions.',
...     labels = ['Cu', 'Ag', 'Au']
... )
```

In the above code, the variable `d1` is an instance of *Dimension* class. Now add this instance to the *add_dimension()* method.

```
>>> new_data.add_dimension(d1)
```

This generates a 3D{0} dataset with the data structure —

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dimension dataset",
    "dimensions": [
      {
        "type": "linear",
        "description": "This is a linear dimension",
        "count": 10,
        "increment": "0.1 s",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
      },
      {
        "type": "monotonic",
        "description": "This is a monotonic dimension",
        "coordinates": [
          "1 µG",
          "2.1 mG",
          "12.4 G",
          "0.5 T",
          "2 T"
        ],
        "quantity_name": "magnetic flux density"
      },
      {
        "type": "labeled",
        "description": "This is a labeled dimensions.",
        "labels": [
          "Cu",
          "Ag",
          "Au"
        ]
      }
    ],
    "dependent_variables": []
  }
}
```

Attention: When using a *Dimension* instance as an argument of the *add_dimension()* method, one must be aware that instances in Python are passed by reference. Therefore, any changes to the instance `d1`, in the above example, will affect the corresponding dimension instance in the `new_data` instance. To be safe, as a general recommendation, one should always pass a copy of the instance to the *add_dimension()* method. We allow the use of *Dimension* objects as arguments because it provides an easy alternative for copying an instance of the *Dimension* class from one *CSDM* instance to another.

5.3 How to add instances of DependentVariable class

In the previous two sections, we illustrated, creating a new dataset and adding dimensions to the dataset. In this example, we illustrate adding dependent variables to the dataset. Let's start by creating a new dataset,

```
>>> import csdmpy as cp
>>> new_data = cp.new(description='A new test dependent variables dataset')
```

An instance of the DependentVariable class is added using the *add_dependent_variable()* method of the *CSDM* instance. There are two subtypes of DependentVariable class:

- **InternalDependentVariable**: We refer to an instance of the DependentVariable as *internal* when the components of the dependent variable are listed along with the other metadata specifying the dependent variable.
- **ExternalDependentVariable**: We refer to an instance of the DependentVariable as *external* when the components of the dependent variable are stored in an external file as binary data either locally or at a remote server.

In version 1.0, the CSD model allows dependent variables with quantity types as

- scalar
- vector_n
- pixel_n
- matrix_n_m
- symmetric_matrix_n

See also:

Read more about *DependentVariable API*.

5.3.1 Adding a scalar dependent variables

A dependent variable with quantity type *scalar* is a single-component dependent variable. Consider the following python dictionary

```
>>> d0 = {
...     'type': 'internal',
...     'quantity_type': 'scalar',
...     'description': 'This is an internal scalar dependent variable',
...     'unit': 'cm',
...     'components': [np.arange(100)]
... }
```

Here, the components are scalar physical quantities with the unit *cm*. The components are listed as the value of the *components* keyword, and therefore, the value of the *type* keyword is specified as *internal*.

Note: The value of the components attribute is a list of numpy array. In csdmpy, the first dimension is reserved for the components. Since this example has only one component, we specify it as a list of numpy array. Alternatively, one could also assign `np.arange(100).reshape(np.newaxis, 100)` as the value of the components attribute.

To add a dependent variable to the `new_data` instance, use the *add_dependent_variable()* method as

```
>>> new_data.add_dependent_variable(d0)
```

This will generate and add a *DependentVariable* object to the list of dependent variables, thereby creating a 0D{1} dataset. The data structure, after adding the dependent variable is,

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dependent variables dataset",
    "dimensions": [],
    "dependent_variables": [
      {
        "type": "internal",
        "description": "This is an internal scalar dependent variable",
        "unit": "cm",
        "quantity_name": "length",
        "numeric_type": "int64",
        "quantity_type": "scalar",
        "components": [
          [
            "0, 1, ..., 98, 99"
          ]
        ]
      }
    ]
  }
}
```

5.3.2 Adding a multi-component dependent variables

In this next example, we demonstrate how to add a dependent variable with multiple components. This time we use keywords as the argument of the *add_dependent_variable()* method to add a new dependent variable.

```
>>> new_data.add_dependent_variable(
...     type='internal',
...     description='This is an internal vector dependent variable',
...     quantity_type='vector_3',
...     unit='kg * m / s^2',
...     components=np.arange(300, dtype='complex64').reshape(3,100)
... )
```

Notice, the value of the *components* keyword is a numpy array of shape 3 x 100, where 3 is the number of components, and 100 is the number of points per component. Here we specify *vector_3* as the value of the *quantity_type*, indicating that the three components of the dependent variable should be interpreted as vector dataset. Users may, however, substitute this value with any valid *quantity_type* consistent with the number of components. For example, the above dataset may also be interpreted as image data if the quantity type is given as *pixel_3*.

The data structure after adding the above dependent variable is

```
>>> print(new_data.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "A new test dependent variables dataset",
    "dimensions": [],
    "dependent_variables": [
      {
```

(continues on next page)

(continued from previous page)

```
        "type": "internal",
        "description": "This is an internal scalar dependent variable",
        "unit": "cm",
        "quantity_name": "length",
        "numeric_type": "int64",
        "quantity_type": "scalar",
        "components": [
          [
            "0, 1, ..., 98, 99"
          ]
        ]
      },
      {
        "type": "internal",
        "description": "This is an internal vector dependent variable",
        "unit": "kg * m * s^-2",
        "quantity_name": "force",
        "numeric_type": "complex64",
        "quantity_type": "vector_3",
        "components": [
          [
            "0j, (1+0j), ..., (98+0j), (99+0j)"
          ],
          [
            "(100+0j), (101+0j), ..., (198+0j), (199+0j)"
          ],
          [
            "(200+0j), (201+0j), ..., (298+0j), (299+0j)"
          ]
        ]
      }
    ]
  }
}
```

5.4 How to save datasets

An instance of a *CSDM* object is serialized as a csdf/csdfe JSON-format file with the `save()` method. When serializing the dependent-variable from the CSDM object to the data-file, the *csdmpy* module uses the value of the dependent variable's `encoding` attribute to determine the encoding type of the serialized data. There are three encoding types for the dependent variables:

- none
- base64
- raw

Note: By default, all instances of `DependentVariable` from a `CSDM` object are serialized as base64 strings.

For the following examples, consider `data` as an instance of the `CSDM` class.

5.4.1 Serializing dependent-variables

To serialize a dependent variable with a given encoding type, set the value of it's encoding attribute to the respective encoding. For example,

As "none" encoding

```
>>> data.dependent_variables[0].encoding = "none"
>>> data.save('my_file.csdf')
```

The above code will serialize the dependent variable at index zero to a JSON file, *my_file.csdf*, where each component of the dependent variable is serialized as an array of JSON number.

As "base64" encoding

```
>>> data.dependent_variables[0].encoding = "base64"
>>> data.save('my_file.csdf')
```

The above code will serialize the dependent variable at index zero to a JSON file, *my_file.csdf*, where each component of the dependent variable is serialized as a base64 string.

As "raw" encoding

```
>>> data.dependent_variables[0].encoding = "raw"
>>> data.save('my_file.csdfe')
```

The above code will serialize the metadata from the dependent variable at index zero to a JSON file, *my_file.csdfe*, which includes a link to an external file where the components of the respective dependent variable are serialized as a binary array. The binary file is named, *my_file_0.dat*, where *my_file* is the filename from the argument of the save method, and *0* is the index number of the dependent variable from the CSDM object.

Multiple encoding types

In the case of multiple dependent-variables, you may choose to serialize each dependent variables with a different encoding, for example,

```
>>> my_data.dependent_variables[0].encoding = "raw"
>>> my_data.dependent_variables[1].encoding = "base64"
>>> my_data.dependent_variables[2].encoding = "none"
>>> my_data.dependent_variables[3].encoding = "base64"
>>> my_data.save('my_file.csdfe')
```

In the above example, `my_data` is a CSDM object containing four *`DependentVariable`* objects. Here, we serialize the dependent variable at index two with `none`, the dependent variables at index one and three with `bae64`, and the dependent variables at index zero with `raw` encoding, respectively.

Note: Because an instance of the dependent variable, that is, the index zero in the above example, is set to be serialized with an external subtype, the corresponding file should be saved with a .csdfe extension.

5.5 An emoji 🤷 example

Let's make use of what we learned so far and create a simple 1D{1} dataset. To make it interesting, let's create an emoji dataset.

Start by importing the *csdmpy* package.

```
>>> import csdmpy as cp
```

Create a new dataset with the *new()* method.

```
>>> fundata = cp.new(description='An emoji dataset')
```

Here, *fundata* is an instance of the *CSDM* class with a 0D{0} dataset. The data structure of this instance is

```
>>> print(fundata.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "An emoji dataset",
    "dimensions": [],
    "dependent_variables": []
  }
}
```

Add a labeled dimension to the *fundata* instance. Here, we'll make use of python dictionary.

```
>>> x = dict(type='labeled', labels=['🤷','🤷','🤷','🤷','🤷','🤷'])
```

The above python dictionary contains two keys. The *type* key identifies the dimension as a labeled dimension while the *labels* key holds an array of labels. In this example, the labels are emojis. Add this dictionary as an argument of the *add_dimension()* method of the *fundata* instance.

```
>>> fundata.add_dimension(x)
>>> print(fundata.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "An emoji dataset",
    "dimensions": [
      {
        "type": "labeled",
        "labels": [
          "🤷",
          "🤷",
          "🤷",
          "🤷",
          "🤷",
          "🤷"
        ]
      }
    ],
    "dependent_variables": []
  }
}
```

We have successfully added a labeled dimension to the *fundata* instance.

Next, add a dependent variable. Set up a python dictionary corresponding to the dependent variable object and add this dictionary as an argument of the *add_dependent_variable()* method of the *fundata* instance.

```
>>> y =dict(type='internal', numeric_type='float32', quantity_type='scalar',
...     components=[[0.5, 0.25, 1, 2, 1, 0.25]])
>>> fundata.add_dependent_variable(y)
```

Here, the python dictionary contains *type*, *numeric_type*, and *components* key. The value of the *components* key holds an array of data values corresponding to the labels from the labeled dimension.

Now, we have a dataset…

```
>>> print(fundata.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "An emoji dataset",
    "dimensions": [
      {
        "type": "labeled",
        "labels": [
          "",
          "",
          "",
          "",
          "",
          ""
        ]
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "numeric_type": "float32",
        "quantity_type": "scalar",
        "components": [
          [
            "0.5, 0.25, ..., 1.0, 0.25"
          ]
        ]
      }
    ]
  }
}
```

To serialize this file, use the *save()* method of the *fundata* instance as

```
>>> fundata.dependent_variables[0].encoding = 'base64'
>>> fundata.save('my_file.csdf')
```

In the above code, the components from the *dependent_variables* attribute at index zero, are encoded as *base64* strings before serializing to the *my_file.csdf* file.

You may also save the components as a binary file, in which case, the file is serialized with a *.csdfe* file extension.

```
>>> fundata.dependent_variables[0].encoding = 'raw'
>>> fundata.save('my_file_raw.csdfe')
```

CHAPTER

SIX

API-REFERENCE

6.1 csdmpy

The *csdmpy* is a python package for importing and exporting files serialized with the core scientific dataset model file-format. The package supports a p-component dependent variable, $\mathbf{U} \equiv \{\mathbf{U}_0, \ldots, \mathbf{U}_q, \ldots, \mathbf{U}_{p-1}\}$, which is discretely sampled at M unique points in a d-dimensional space $(\mathbf{X}_0, \ldots \mathbf{X}_k, \ldots \mathbf{X}_{d-1})$. Besides, the package also supports multiple dependent variables, $\mathbf{U}_i$, sharing the same d-dimensional space.

Here, every dataset is an instance of the *CSDM* class, which holds a list of dimensions and dependent variables. Every dimension, $\mathbf{X}_k$, is an instance of the *Dimension* class, while every dependent variable, $\mathbf{U}_i$, is an instance of the *DependentVariable* class.

6.1.1 Methods

Methods Summary

load([filename, application])	Load a .csdf/.csdfe file and return an instance of *CSDM* class.
new([description])	Create a new instance of the *CSDM* class containing a 0D{0} dataset.
parse_dict(dictionary)	Parse a CSDM compliant python dictionary and return a CSDM object.
loads(string)	Loads a JSON serialized string as a CSDM object.
plot(data_object, **kwargs)	Helper function for plotting basic 1D and 2D datasets.

Method Documentation

csdmpy.**load**(*filename=None*, *application=False*)

Load a .csdf/.csdfe file and return an instance of *CSDM* class.

The file must be a JSON serialization of the CSD Model.

Example

```
>>> data1 = cp.load('local_address/file.csdf')
>>> data2 = cp.load('url_address/file.csdf')
```

Parameters

- **filename** (*str*) – A local or remote address to the *.csdf or '.csdfe* file.
- **application** (*bool*) – If true, the application metadata from application that last serialized the file will be imported. Default is False.
- **sort_fft_order** (*bool*) – If true, the coordinates and the components corresponding to the dimension with *complex_fft* as True will be sorted upon import and the corresponding *complex_fft* key-value will be set to False. Default is True.

Returns A CSDM instance.

csdmpy.**new**(*description=''*)

Create a new instance of the *CSDM* class containing a 0D{0} dataset.

Parameters **description** (*str*) – A string describing the the csdm object. This is optional.

Example

```
>>> import csdmpy as cp
>>> emptydata = cp.new(description='Testing Testing 1 2 3')
>>> print(emptydata.data_structure)
{
  "csdm": {
    "version": "1.0",
    "description": "Testing Testing 1 2 3",
    "dimensions": [],
    "dependent_variables": []
  }
}
```

Returns A CSDM instance.

csdmpy.**parse_dict**(*dictionary*)

Parse a CSDM compliant python dictionary and return a CSDM object.

Parameters **dictionary** – A CSDM compliant python dictionary.

csdmpy.**loads**(*string*)

Loads a JSON serialized string as a CSDM object.

Parameters **string** – A JSON serialized CSDM string.

Returns A CSDM object

Example

```
>>> object_from_string = cp.loads(cp.new('A test dump').dumps())
>>> print(object_from_string.data_structure)
{
  "csdm": {
    "version": "1.0",
    "timestamp": "2019-10-21T20:33:17Z",
    "description": "A test dump",
    "dimensions": [],
    "dependent_variables": []
  }
}
```

csdmpy.**plot**(*data_object*, ***kwargs*)

Helper function for plotting basic 1D and 2D datasets.

Parameters

- **data_object** – The csdm object
- **reverse_axis** – An array of boolean specifying which dimensions will be displayed in reverse.
- **kwargs** – Additional kwargs which are passed to the matplotlib plot or imshow functions.

Example

```
>>> cp.plot(data_object)
```

6.2 CSDM

class csdmpy.csdm.**CSDM**(*filename=''*, *version=None*, *description=''*, ***kwargs*)

Bases: object

Create an instance of a CSDM class.

This class is based on the root CSDM object of the core scientific dataset (CSD) model. The class is a composition of the *DependentVariable* and *Dimension* instances, where an instance of the *DependentVariable* class describes a p-component dependent variable, and an instance of the *Dimension* class describes a dimension of a d-dimensional space. Additional attributes of this class are listed below.

Attributes Summary

version	Version number of the CSD model on file.
description	Description of the dataset.
read_only	If True, the data-file is serialized as read only, otherwise, False.
timestamp	Timestamp from when the file was last serialized.
geographic_coordinate	Geographic coordinate, if present, from where the file was last serialized.

continues on next page

Table 2 – continued from previous page

dimensions	Tuple of *Dimension* instances.
dependent_variables	Tuple of *DependentVariable* instances.
tags	List of tags attached to the dataset.
application	Application metadata dictionary of the CSDM object.
data_structure	Json serialized string describing the CSDM class instance.
filename	Local file address of the current file.

Methods Summary

add_dimension(*args, **kwargs)	Add a new *Dimension* instance to the *CSDM* instance.
add_dependent_variable(*args, **kwargs)	Add a new *DependentVariable* instance to the *CSDM* instance.
to_dict([update_timestamp, read_only, version])	Serialize the *CSDM* instance as a python dictionary.
dumps([update_timestamp, read_only, version])	Serialize the *CSDM* instance as a JSON data-exchange string.
save([filename, read_only, version, …])	Serialize the *CSDM* instance as a JSON data-exchange file.
copy()	Create a copy of the current CSDM instance.

Attributes Documentation

version
Version number of the CSD model on file.

description
Description of the dataset.

The default value is an empty string, ‘’.

Example

```
>>> print(data.description)
A simulated sine curve.
```

Returns A string of UTF-8 allows characters describing the dataset.

Raises **TypeError** – When the assigned value is not a string.

read_only
If True, the data-file is serialized as read only, otherwise, False.

By default, the *CSDM* object loads a copy of the .csdf(e) file, irrespective of the value of the *read_only* attribute. The value of this attribute may be toggled at any time after the file import. When serializing the *.csdf(e)* file, if the value of the *read_only* attribute is found True, the file will be serialized as read only.

timestamp
Timestamp from when the file was last serialized.

The timestamp stamp is a string representation of the Coordinated Universal Time (UTC) formatted according to the iso-8601 standard.

Raises **AttributeError** – When the attribute is modified.

geographic_coordinate
Geographic coordinate, if present, from where the file was last serialized.

The geographic coordinates correspond to the location where the file was last serialized. If present, the geographic coordinates are described with three attributes, the required latitude and longitude, and an optional altitude.

Raises **AttributeError** – When the attribute is modified.

dimensions
Tuple of *Dimension* instances.

dependent_variables
Tuple of *DependentVariable* instances.

tags
List of tags attached to the dataset.

application
Application metadata dictionary of the CSDM object.

```
>>> print(data.application)
{}
```

By default, the application attribute is an empty dictionary, that is, the application metadata stored by the previous application is ignored upon file import.

The application metadata may, however, be retained with a request via the *load()* method. This feature may be useful to related applications where application metadata might contain additional information. The attribute may be updated with a python dictionary.

The application attribute is where an application can place its own metadata as a python dictionary object containing application specific metadata, using a reverse domain name notation string as the attribute key, for example,

Example

```
>>> data.application = {
...     "com.example.myApp" : {
...         "myApp_key": "myApp_metadata"
...      }
... }
>>> print(data.application)
{'com.example.myApp': {'myApp_key': 'myApp_metadata'}}
```

Returns Python dictionary object with the application metadata.

data_structure
Json serialized string describing the CSDM class instance.

The data_structure attribute is only intended for a quick preview of the dataset. This JSON serialized string from this attribute avoids displaying large datasets. Do not use the value of this attribute to save the data to a file, instead use the `save()` methods of the instance.

Raises **AttributeError** – When modified.

filename
Local file address of the current file.

Methods Documentation

add_dimension(**args*, ***kwargs*)
Add a new *Dimension* instance to the *CSDM* instance.

There are three ways to add a new independent variable.

From a python dictionary containing valid keywords.

```
>>> import csdmpy as cp
>>> datamodel = cp.new()
>>> py_dictionary = {
...     'type': 'linear',
...     'increment': '5 G',
...     'count': 50,
...     'coordinates_offset': '-10 mT'
... }
>>> datamodel.add_dimension(py_dictionary)
```

From a list of valid keyword arguments.

```
>>> datamodel.add_dimension(
...     type = 'linear',
...     increment = '5 G',
...     count = 50,
...     coordinates_offset = '-10 mT'
... )
```

From an Dimension instance.

```
>>> from csdmpy import Dimension
>>> datamodel = cp.new()
>>> var1 = Dimension(type = 'linear',
...                  increment = '5 G',
...                  count = 50,
...                  coordinates_offset = '-10 mT')
>>> datamodel.add_dimension(var1)
>>> print(datamodel.data_structure)
{
  "csdm": {
    "version": "1.0",
    "dimensions": [
      {
        "type": "linear",
        "count": 50,
        "increment": "5.0 G",
        "coordinates_offset": "-10.0 mT",
        "quantity_name": "magnetic flux density"
      }
    ],
    "dependent_variables": []
  }
}
```

For the last method, the instance `var1` is added to the `datamodel` as a reference, *i.e.*, if the instance `var1` is destroyed, the `datamodel` instance will become corrupt. As a recommendation always pass a copy of the *Dimension* instance to the *`add_dimension()`* method. We provide the later alternative for it is useful for copying an *Dimension* instance from one *CSDM* instance to another.

`add_dependent_variable`(**args*, ***kwargs*)
Add a new *DependentVariable* instance to the *CSDM* instance.

There are again three ways to add a new dependent variable instance.

From a python dictionary containing valid keywords.

```
>>> import numpy as np

>>> datamodel = cp.new()

>>> numpy_array = (100*np.random.rand(3,50)).astype(np.uint8)
>>> py_dictionary = {
...     'type': 'internal',
...     'components': numpy_array,
...     'name': 'star',
...     'unit': 'W s',
...     'quantity_name': 'energy',
...     'quantity_type': 'pixel_3'
... }
>>> datamodel.add_dependent_variable(py_dictionary)
```

From a list of valid keyword arguments.

```
>>> datamodel.add_dependent_variable(type='internal',
...                                  name='star',
...                                  unit='W s',
...                                  quantity_type='pixel_3',
...                                  components=numpy_array)
```

From a DependentVariable instance.

```
>>> from csdmpy import DependentVariable
>>> var1 = DependentVariable(type='internal',
...                          name='star',
...                          unit='W s',
...                          quantity_type='pixel_3',
...                          components=numpy_array)
>>> datamodel.add_dependent_variable(var1)
```

If passing a *DependentVariable* instance, as a general recommendation, always pass a copy of the DependentVariable instance to the *`add_dependent_variable()`* method. We provide the later alternative as it is useful for copying a DependentVariable instance from one *CSDM* instance to another.

`to_dict`(*update_timestamp=False*, *read_only=False*, *version='1.0'*)
Serialize the *CSDM* instance as a python dictionary.

Parameters

- **`update_timestamp`** (*`bool`*) – If True, timestamp is updated to current time.
- **`read_only`** (*`bool`*) – If true, the read_only flag is set true.
- **`version`** (*`str`*) – Serialize the dict with the given csdm version.

Example

```
>>> data.to_dict()
{'csdm': {'version': '1.0', 'timestamp': '1994-11-05T13:15:30Z', 'geographic_
↪coordinate': {'latitude': '10 deg', 'longitude': '93.2 deg', 'altitude':
↪'10 m'}, 'description': 'A simulated sine curve.', 'dimensions': [{'type':
↪'linear', 'description': 'A temporal dimension.', 'count': 10, 'increment':
↪'0.1 s', 'quantity_name': 'time', 'label': 'time', 'reciprocal': {'quantity_
↪name': 'frequency'}}], 'dependent_variables': [{'type': 'internal',
↪'description': 'A response dependent variable.', 'name': 'sine curve',
↪'encoding': 'base64', 'numeric_type': 'float32', 'quantity_type': 'scalar',
↪'component_labels': ['response'], 'components': ['AAAAABh5Fj9xeHM/
↪cXhzPxh5Fj8yMQ0lGHkWv3F4c79xeHO/GHkWvw==']}]}}
```

dumps (*update_timestamp=False*, *read_only=False*, *version='1.0'*)
Serialize the *CSDM* instance as a JSON data-exchange string.

Parameters

- **update_timestamp** (*bool*) – If True, timestamp is updated to current time.
- **read_only** (*bool*) – If true, the file is serialized as read_only.
- **version** (*str*) – The file is serialized with the given CSD model version.

Example

```
>>> data.dumps()
```

save (*filename=''*, *read_only=False*, *version='1.0'*, *output_device=None*)
Serialize the *CSDM* instance as a JSON data-exchange file.

There are two types of file serialization extensions, *.csdf* and *.csdfe*. In the CSD model, when every instance of the DependentVariable objects from a CSDM class has an *internal* subtype, the corresponding CSDM instance is serialized with a *.csdf* file extension. If any single DependentVariable instance has an *external* subtype, the CSDM instance is serialized with a *.csdfe* file extension. The two different file extensions are used to alert the end-user of the possible deserialization error associated with the *.csdfe* file extensions had the external data file becomes inaccessible.

In *csdmpy*, however, irrespective of the dependent variable subtypes from the serialized JSON file, by default, all instances of DependentVariable class are treated an *internal* after import. Therefore, when serialized, the CSDM object should be stored as a *.csdf* file.

To store a file as a *.csdfe* file, the user much set the value of the `encoding` attribute from the dependent variables to `raw`. In which case, a binary file named *filename_i.dat* will be generated where i is the i^{th} dependent variable. The parameter *filename* is an argument of this method.

Note: Only dependent variables with `encoding="raw"` will be serialized to a binary file.

Parameters

- **filename** (*str*) – The filename of the serialized file.
- **read_only** (*bool*) – If true, the file is serialized as read_only.
- **version** (*str*) – The file is serialized with the given CSD model version.

- **output_device** (*object*) – Object where the data is written. If provided, the argument *filename* become irrelevant.

Example

```
>>> data.save('my_file.csdf')
```

copy()
Create a copy of the current CSDM instance.

Example

```
>>> data.copy()
```

Returns A CSDM instance.

6.3 Dimension

class csdmpy.dimensions.**Dimension**(**args*, ***kwargs*)
Bases: object

Create an instance of the Dimension class.

An instance of this class describes a dimension of a multi-dimensional system. In version 1.0 of the CSD model, there are three subtypes of the Dimension class:

- linearDimension_uml,
- monotonicDimension_uml, and
- labeledDimension_uml.

Creating an instance of a dimension object

There are two ways of creating a new instance of a Dimension class.

From a python dictionary containing valid keywords.

```
>>> from csdmpy import Dimension
>>> dimension_dictionary = {
...     'type': 'linear',
...     'description': 'test',
...     'increment': '5 G',
...     'count': 10,
...     'coordinates_offset': '10 mT',
...     'origin_offset': '10 T'
... }
>>> x = Dimension(dimension_dictionary)
```

Here, *dimension_dictionary* is the python dictionary.

From valid keyword arguments.

```
>>> x = Dimension(type = 'linear',
...               description = 'test',
...               increment = '5 G',
...               count = 10,
...               coordinates_offset = '10 mT',
...               origin_offset = '10 T')
```

Attributes Summary

type	The dimension subtype.
description	Brief description of the dimension object.
application	Application metadata dictionary of the dimension object.
coordinates	Coordinates, $\mathbf{X}_k$, along the dimension.
absolute_coordinates	Absolute coordinates, $\mathbf{X}_\mathbf{k}^{\text{abs}}$, along the dimension.
count	Number of coordinates, $N_k \geq 1$, along the dimension.
increment	Increment along a *linear* dimension.
coordinates_offset	Offset corresponding to the zero of the indexes array, $\mathbf{J}_k$.
origin_offset	Origin offset, o_k, along the dimension.
complex_fft	Boolean specifying if the coordinates along the dimension are the output of a complex fft.
quantity_name	Quantity name associated with the physical quantities specifying the dimension.
label	Label associated with the dimension.
labels	Ordered list of labels along the *Labeled* dimension.
period	Period of the dimension.
axis_label	Formatted string for displaying label along the dimension axis.
data_structure	Json serialized string describing the Dimension class instance.

Methods Summary

to([unit, equivalencies])	Convert the coordinates along the dimension to the unit, *unit*.
to_dict()	Return Dimension object as a python dictionary.
is_quantitative()	Return True if the dependent variable is quantitative.

Attributes Documentation

type
The dimension subtype.

There are three *valid* subtypes of Dimension class. The valid literals are given by the *DimObjectSubtype* enumeration.

```
>>> print(x.type)
linear
```

Returns A string with a valid dimension subtype.

Raises **AttributeError** – When the attribute is modified.

description
Brief description of the dimension object.

The default value is an empty string, ''. The attribute may be modified, for example,

```
>>> print(x.description)
This is a test

>>> x.description = 'This is a test dimension.'
```

Returns A string of UTF-8 allows characters describing the dimension.

Raises **TypeError** – When the assigned value is not a string.

application
Application metadata dictionary of the dimension object.

```
>>> print(x.application)
{}
```

The application attribute is where an application can place its own metadata as a python dictionary object containing application specific metadata, using a reverse domain name notation string as the attribute key, for example,

```
>>> x.application = {
...     "com.example.myApp" : {
...         "myApp_key": "myApp_metadata"
...      }
... }
>>> print(x.application)
{'com.example.myApp': {'myApp_key': 'myApp_metadata'}}
```

Returns A python dictionary containing dimension application metadata.

coordinates
Coordinates, $\mathbf{X}_k$, along the dimension.

Example

```
>>> print(x.coordinates)
[100. 105. 110. 115. 120. 125. 130. 135. 140. 145.] G
```

For *linear* dimensions, the order of the *coordinates* also depend on the value of the *complex_fft* attributes. For examples, when the value of the *complex_fft* attribute is True, the coordinates are

```
>>> x.complex_fft = True
>>> print(x.coordinates)
[ 75.  80.  85.  90.  95. 100. 105. 110. 115. 120.] G
```

Returns A Quantity array of coordinates for quantitative dimensions, *i.e. linear* and *monotonic*.

Returns A Numpy array for labeled dimensions.

Raises **AttributeError** – For dimensions with subtype *linear*.

absolute_coordinates

Absolute coordinates, $\mathbf{X}_\mathbf{k}^{\text{abs}}$, along the dimension.

This attribute is only *valid* for quantitative dimensions, that is, *linear* and *monotonic* dimensions. The absolute coordinates are given as

$$\mathbf{X}_k^{\text{abs}} = \mathbf{X}_k + o_k \mathbf{1}$$

where $\mathbf{X}_k$ are the coordinates along the dimension and o_k is the *origin_offset*. For example, consider

```
>>> print(x.origin_offset)
10.0 T
>>> print(x.coordinates[:5])
[100. 105. 110. 115. 120.] G
```

then the absolute coordinates are

```
>>> print(x.absolute_coordinates[:5])
[100100. 100105. 100110. 100115. 100120.] G
```

For *linear* dimensions, the order of the *absolute_coordinates* further depend on the value of the *complex_fft* attributes. For examples, when the value of the *complex_fft* attribute is True, the absolute coordinates are

```
>>> x.complex_fft = True
>>> print(x.absolute_coordinates[:5])
[100075. 100080. 100085. 100090. 100095.] G
```

Returns A Quantity array of absolute coordinates for quantitative dimensions, *i.e linear* and *monotonic*.

Raises **AttributeError** – For labeled dimensions.

count

Number of coordinates, $N_k \geq 1$, along the dimension.

Example

```
>>> print(x.count)
10
>>> x.count = 5
```

Returns An Integer specifying the number of coordinates along the dimension.

Raises **TypeError** – When the assigned value is not an integer.

increment

Increment along a *linear* dimension.

The attribute is only *valid* for Dimension instances with the subtype *linear*. When assigning a value, the dimensionality of the value must be consistent with the dimensionality of other members specifying the dimension.

Example

```
>>> print(x.increment)
5.0 G
>>> x.increment = "0.1 G"
>>> print(x.coordinates)
[100.  100.1 100.2 100.3 100.4 100.5 100.6 100.7 100.8 100.9] G
```

Returns A Quantity instance with the increment along the dimension.

Raises

- **AttributeError** – For dimension with subtypes other than *linear*.
- **TypeError** – When the assigned value is not a string containing a quantity or a Quantity object.

coordinates_offset

Offset corresponding to the zero of the indexes array, $\mathbf{J}_k$.

When assigning a value, the dimensionality of the value must be consistent with the dimensionality of the other members specifying the dimension.

Example

```
>>> print(x.coordinates_offset)
10.0 mT
>>> x.coordinates_offset = "0 T"
>>> print(x.coordinates)
[ 0.  5. 10. 15. 20. 25. 30. 35. 40. 45.] G
```

The attribute is *invalid* for *labeled* dimensions.

Returns A Quantity instance with the coordinates offset.

Raises

- **AttributeError** – For *labeled* dimensions.

- **TypeError** – When the assigned value is not a string containing a quantity or a Quantity object.

origin_offset

Origin offset, o_k, along the dimension.

When assigning a value, the dimensionality of the value must be consistent with the dimensionality of other members specifying the dimension.

Example

```
>>> print(x.origin_offset)
10.0 T
>>> x.origin_offset = "1e5 G"
```

The origin offset only affect the absolute_coordinates along the dimension. This attribute is *invalid* for *labeled* dimensions.

Returns A Quantity instance with the origin offset.

Raises

- **AttributeError** – For *labeled* dimensions.
- **TypeError** – When the assigned value is not a string containing a quantity or a Quantity object.

complex_fft

Boolean specifying if the coordinates along the dimension are the output of a complex fft.

This attribute is only *valid* for the Dimension instances with *linear* subtype. The value of this attribute is a boolean specifying if the coordinates along the dimension are evaluated as the output of a complex fast Fourier transform (FFT) routine. For example, consider the following Dimension object,

```
>>> test = Dimension(
...             type='linear',
...             increment = '1',
...             count = 10
...         )

>>> test.complex_fft
False
>>> print(test.coordinates)
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]

>>> test.complex_fft = True
>>> print(test.coordinates)
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

Returns A Boolean.

Raises **TypeError** – When the assigned value is not a boolean.

quantity_name

Quantity name associated with the physical quantities specifying the dimension.

The attribute is *invalid* for the labeled dimension.

```
>>> print(x.quantity_name)
magnetic flux density
```

Returns A string with the *quantity name*.

Raises

- **AttributeError** – For *labeled* dimensions.
- **NotImplementedError** – When assigning a value.

label

Label associated with the dimension.

Example

```
>>> print(x.label)
field strength
>>> x.label = 'magnetic field strength'
```

Returns A string containing the label.

Raises **TypeError** – When the assigned value is not a string.

labels

Ordered list of labels along the *Labeled* dimension.

Consider the following labeled dimension,

```
>>> x2 = Dimension(
...         type='labeled',
...         labels=['Cu', 'Ag', 'Au']
...       )
```

then the lables along the labeled dimension are

```
>>> print(x2.labels)
['Cu' 'Ag' 'Au']
```

Note: For Labeled dimension, the *coordinates* attribute is an alias of *labels* attribute. For example,

```
>>> np.all(x2.coordinates == x2.labels)
True
```

In the above example, x2 is an instance of the *Dimension* class with *labeled* subtype.

Returns A Numpy array with labels along the dimension.

Raises **AttributeError** – For dimensions with subtype other than *labeled*.

period

Period of the dimension.

The default value of the period is infinity, i.e., the dimension is non-periodic.

Example

```
>>> print(x.period)
inf G
>>> x.period = '1 T'
```

To assign a dimension as non-periodic, one of the following may be used,

```
>>> x.period = '1/0 T'
>>> x.period = 'infinity μT'
>>> x.period = '∞ G'
```

Attention: The physical quantity of the period must be consistent with other physical quantities specifying the dimension.

Returns A Quantity instance with the period of the dimension.

Raises

- **AttributeError** – For *labeled* dimensions.
- **TypeError** – When the assigned value is not a string containing a quantity or a Quantity object.

axis_label

Formatted string for displaying label along the dimension axis.

This attribute is not a part of the original core scientific dataset model, however, it is a convenient supplementary attribute that provides a formated string ready for labeling dimension axes. For quantitative dimensions, this attributes returns a string, *label / unit*, if the *label* is a non-empty string, otherwise, *quantity_name / unit*. Here *quantity_name* and *label* are the attributes of the *Dimension* instances, and *unit* is the unit associated with the coordinates along the dimension. For examples,

```
>>> x.label
'field strength'
>>> x.axis_label
'field strength / (G)'
```

For *labeled* dimensions, this attribute returns *label*.

Returns A formated string of label.

Raises **AttributeError** – When assigned a value.

data_structure

Json serialized string describing the Dimension class instance.

This supplementary attribute is useful for a quick preview of the dimension object. The attribute cannot be modified.

```
>>> print(x.data_structure)
{
  "type": "linear",
  "description": "This is a test",
  "count": 10,
  "increment": "5.0 G",
  "coordinates_offset": "10.0 mT",
```

(continues on next page)

(continued from previous page)

```
  "origin_offset": "10.0 T",
  "quantity_name": "magnetic flux density",
  "label": "field strength"
}
```

Returns A json serialized string of the dimension object.

Raises **AttributeError** – When modified.

Method Documentation

to(*unit=''*, *equivalencies=None*)

Convert the coordinates along the dimension to the unit, *unit*.

This method is a wrapper of the *to* method from the Quantity class and is only *valid* for physical dimensions.

Example

```
>>> print(x.coordinates)
[100. 105. 110. 115. 120. 125. 130. 135. 140. 145.] G
>>> x.to('mT')
>>> print(x.coordinates)
[10.  10.5 11.  11.5 12.  12.5 13.  13.5 14.  14.5] mT
```

Parameters **unit** – A string containing a unit with the same dimensionality as the coordinates along the dimension.

Raises **AttributeError** – For *labeled* dimensions.

to_dict()

Return Dimension object as a python dictionary.

Example

```
>>> x.to_dict()
{'type': 'linear', 'description': 'This is a test', 'count': 10, 'increment':
↪'5.0 G', 'coordinates_offset': '10.0 mT', 'origin_offset': '10.0 T',
↪'quantity_name': 'magnetic flux density', 'label': 'field strength'}
```

is_quantitative()

Return True if the dependent variable is quantitative.

Example

```
>>> x.is_quantitative()
True
```

6.4 DependentVariable

class csdmpy.dependent_variables.**DependentVariable**(**args*, ***kwargs*)

Bases: object

Create an instance of the DependentVariable class.

The instance of this class represents a dependent variable, **U**. A dependent variable holds p-component data values, where $p > 0$ is an integer. For example, a scalar is single-component ($p = 1$), a vector may have up to n-components ($p = n$), while a second rank symmetric tensor have six unique component ($p = 6$).

Creating a new dependent variable.

There are two ways of creating a new instance of a DependentVariable class.

From a python dictionary containing valid keywords.

```
>>> from csdmpy import DependentVariable
>>> import numpy as np
>>> numpy_array = np.arange(30).reshape(3,10).astype(np.float32)

>>> dependent_variable_dictionary = {
...     'type': 'internal',
...     'components': numpy_array,
...     'name': 'star',
...     'unit': 'W s',
...     'quantity_name': 'energy',
...     'quantity_type': 'pixel_3'
... }
>>> y = DependentVariable(dependent_variable_dictionary)
```

Here, *dependent_variable_dictionary* is the python dictionary.

From valid keyword arguments.

```
>>> y = DependentVariable(
...         type='internal',
...         name='star',
...         unit='W s',
...         quantity_type='pixel_3',
...         components=numpy_array
...     )
```

Attributes Summary

type	The dependent variable subtype.
description	Brief description of the dependent variables.
application	Application metadata of the DependentVariable object.
name	Name of the dependent variable.
unit	Unit associated with the dependent variable.
quantity_name	Quantity name of the physical quantities associated with the dependent variable.
encoding	The encoding method used in representing the dependent variable.
numeric_type	The numeric type of the data values from the dependent variable.
quantity_type	Quantity type of the dependent variable.
component_labels	List of labels corresponding to the components of the dependent variable.
components	Component array of the dependent variable.
components_url	URL where the data components of the dependent variable are stored.
axis_label	List of formatted string labels for each component of the dependent variable.
data_structure	Json serialized string describing the DependentVariable class instance.

Methods Summary

to(unit)	Convert the unit of the dependent variable to the *unit*.
to_dict()	Return DependentVariable object as a python dictionary.

Attributes Documentation

type

The dependent variable subtype.

There are two *valid* subtypes of DependentVariable class with the following enumeration literals,

```
internal
external
```

corresponding to Internal and External sub class. By default, all instances of the DependentVariable class are assigned as *internal* upon import. The user may update the value of this attribute, at any time, with a string containing a valid *type* literal, for example,

```
>>> print(y.type)
internal
```

(continues on next page)

(continued from previous page)

```
>>> y.type = 'external'
```

When *type* is external, the data values from the corresponding dependent variable are serialized to an external file within the same directory as the *.csdfe* file.

Returns A string with a *valid* dependent variable subtype.

Raises **ValueError** – When an invalid value is assigned.

description

Brief description of the dependent variables.

The default value is an empty string, ''.

```
>>> print(y.description)
A test image
>>> y.description = 'A test pixel_3 image'
>>> print(y.description)
A test pixel_3 image
```

Returns A string of UTF-8 allowed characters describing the dependent variable.

Raises **TypeError** – When the assigned value is not a string.

application

Application metadata of the DependentVariable object.

```
>>> print(y.application)
{}
```

The application attribute is where an application can place its own metadata as a python dictionary object containing the application specific metadata, using a reverse domain name notation string as the attribute key, for example,

```
>>> y.application = {
...     "com.example.myApp" : {
...         "myApp_key": "myApp_metadata"
...      }
... }
>>> print(y.application)
{'com.example.myApp': {'myApp_key': 'myApp_metadata'}}
```

Please refer to the Core Scientific Dataset Model article for details.

Returns A python dictionary containing dependent variable application metadata.

name

Name of the dependent variable.

```
>>> y.name
'star'
>>> y.name = 'rock star'
```

Returns A string containing the name of the dependent variable.

Raises **TypeError** – When the assigned value is not a string.

unit
Unit associated with the dependent variable.

Note: The attribute cannot be modified. To convert the unit, use the *to()* method of the class instance.

```
>>> y.unit
Unit("s W")
```

Returns A *Unit* object from astropy.unit package.

Raises **AttributeError** – When assigned a value.

quantity_name
Quantity name of the physical quantities associated with the dependent variable.

```
>>> y.quantity_name
'energy'
```

Returns A string with the quantity name associated with the dependent variable physical quantities .

Raises **NotImplementedError** – When assigning a value.

encoding
The encoding method used in representing the dependent variable.

The value of this attribute determines the method used when serializing or deserializing the data values to and from the file. Currently, there are three *valid* encoding methods:

```
raw
base64
none
```

A value, *raw*, means that the data values are serialized as binary data. The value, *base64*, implies that the data values are serialized as base64 strings, while, the value *none* refers to text-based serialization.

By default, the encoding attribute of all dependent variable object are set to *base64* after import. The user may update this attribute, at any time, with a string containing a *valid* encoding literal, for example,

```
>>> y.encoding = 'base64'
```

The value of this attribute will be used in serializing the data to the file, when using the *save()* method.

Returns A string with a *valid* encoding type.

Raises **ValueError** – If an invalid encoding value is assigned.

numeric_type
The numeric type of the data values from the dependent variable.

There are currently twelve *valid* numeric types:

uint8	int8	float32	complex64
uint16	int16	float64	complex128
uint32	int32		
uint64	int64		

When assigning a valid value, this attribute updates the *dtype* of the Numpy array from the corresponding *components* attribute. We recommended the use of the numeric type attribute for updating the *dtype* of the Numpy array. For example,

```
>>> y.numeric_type
'float32'

>>> print(y.components)
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]]

>>> y.numeric_type = 'complex64'

>>> print(y.components[:,:5])
[[ 0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j]
 [10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j]
 [20.+0.j 21.+0.j 22.+0.j 23.+0.j 24.+0.j]]
```

Returns A string with a *valid* numeric type.

Raises **ValueError** – If an invalid numeric type value is assigned.

quantity_type

Quantity type of the dependent variable.

There are currently six *valid* quantity types,

```
scalar
vector_n
pixel_n
matrix_n_m
symmetric_matrix_n
```

where *n* and *m* are integers. The value of the attribute is modified with a string containing a *valid* quantity type.

```
>>> y.quantity_type
'pixel_3'
>>> y.quantity_type = 'vector_3'
```

Returns A string with a *valid* quantity type.

Raises **ValueError** – If an invalid value is assigned.

component_labels

List of labels corresponding to the components of the dependent variable.

```
>>> y.component_labels
['', '', '']
```

To update the *component_labels*, assign an array of strings with same number of elements as the number of components.

```
>>> y.component_labels = ['channel 0', 'channel 1', 'channel 2']
```

The individual labels are accessed with proper indexing, for example,

```
>>> y.component_labels[2]
'channel 2'
```

Returns A list of component label strings.

Raises **TypeError** – When the assigned value is not an array of strings.

components

Component array of the dependent variable.

The value of this attribute, $\mathbb{U}$, is a Numpy array of shape $(p \times N_{d-1} \times ...N_1 \times N_0)$ where p is the number of components, and N_k is the number of points from the k^{th} *Dimension* object.

Note: The shape of the components Numpy array, $(p \times N_{d-1} \times ...N_1 \times N_0)$, is reverse the shape of the components array, $(N_0 \times N_1 \times ...N_{d-1} \times p)$, from the CSD model. This is because CSD model utilizes a column-major order to shape the components array relative to the order of the dimension while Numpy utilizes a row-major order.

The dimensionality of this Numpy array is $d + 1$ where d is the number of dimension objects. The zeroth axis with p points is the number of components.

This attribute can only be updated when the shape of the new array is the same as the shape of the components array.

For example,

```
>>> print(y.components.shape)
(3, 10)
>>> y.numeric_type
'float32'
```

is a three-component dependent variable with ten data values per component. The numeric type of the data values, in this example, is *float32*. To update the components array, assign an array of shape (3, 10) to the *components* attribute. In the following example, we assign a Numpy array,

```
>>> y.components = np.linspace(0,256,30, dtype='u1').reshape(3,10)
>>> y.numeric_type
'uint8'
```

Notice, the value of the *numeric_type* attribute is automatically updated based on the *dtype* of the Numpy array. In this case, from a *float32* to *uint8*. In this other example,

```
>>> try:
...     y.components = np.random.rand(1,10).astype('u1')
... except ValueError as e:
```

(continues on next page)

(continued from previous page)

```
...     print(e)
The shape of `ndarray`, `(1, 10)`, is inconsistent
with the shape of the components array, `(3, 10)`.
```

a *ValueError* is raised because the shape of the input array (1, 10) is not consistent with the shape of the components array, (3, 10).

Returns A Numpy array of components.

Raises **ValueError** – When assigning an array whose shape is not consistent with the shape of the components array.

components_url

URL where the data components of the dependent variable are stored.

This attribute is only informative and cannot be modified. Its value is a string containing the local or remote address of the file where the data values are stored. The attribute is only valid for dependent variable with type, *external*.

Returns A string containing the URL.

Raises **AttributeError** – When assigned a value.

axis_label

List of formatted string labels for each component of the dependent variable.

This attribute is not a part of the original core scientific dataset model, however, it is a convenient supplementary attribute that provides formated string ready for labeling the components of the dependent variable. The string at index *i* is formatted as *component_labels[i] / unit* if *component_labels[i]* is a non-empty string, otherwise, *quantity_name / unit*. Here, *quantity_name*, *component_labels*, and *unit'are the attributes of the :ref:'dv_api* instance. For example,

```
>>> y.axis_label
['energy / (s * W)', 'energy / (s * W)', 'energy / (s * W)']
```

Returns A list of formated component label strings.

Raises **AttributeError** – When assigned a value.

data_structure

Json serialized string describing the DependentVariable class instance.

This supplementary attribute is useful for a quick preview of the dependent variable object. For convenience, the values from the *components* attribute are truncated to the first and the last two numbers per component. The *encoding* keyword is also hidden from this view.

```
>>> print(y.data_structure)
{
  "type": "internal",
  "description": "A test image",
  "name": "star",
  "unit": "s * W",
  "quantity_name": "energy",
  "numeric_type": "float32",
  "quantity_type": "pixel_3",
  "components": [
    [
      "0.0, 1.0, ..., 8.0, 9.0"
```

(continues on next page)

(continued from previous page)

```
    ],
    [
      "10.0, 11.0, ..., 18.0, 19.0"
    ],
    [
      "20.0, 21.0, ..., 28.0, 29.0"
    ]
  ]
}
```

Returns A json serialized string of the dependent variable object.

Raises **AttributeError** – When modified.

Method Documentation

to(*unit*)

Convert the unit of the dependent variable to the *unit*.

Parameters **unit** – A string containing a unit with the same dimensionality as the components of the dependent variable.

```
>>> y.unit
Unit("s W")
>>> print(y.components[0,5])
5.0
>>> y.to('mJ')
>>> y.unit
Unit("mJ")
>>> print(y.components[0,5])
5000.0
```

Note: This method is a wrapper of the *to* method from the Quantity class.

to_dict()

Return DependentVariable object as a python dictionary.

Example

```
>>> y.to_dict()
{'type': 'internal', 'description': 'A test image', 'name': 'star', 'unit':
↪'s * W', 'quantity_name': 'energy', 'encoding': 'none', 'numeric_type':
↪'float32', 'quantity_type': 'pixel_3', 'components': [[0.0, 1.0, 2.0, 3.0,␣
↪4.0, 5.0, 6.0, 7.0, 8.0, 9.0], [10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0,␣
↪17.0, 18.0, 19.0], [20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0,␣
↪29.0]]}
```

INDEX

O

P

Q

R

S

T

U

V