csdmpy

A python library for
Core Scientific Dataset Model

Documentation

Release 0.2.1

Deepansh J. Srivastava

May 21, 2020

TABLE OF CONTENTS

CHAPTER

ONE

ABOUT

The *csdmpy* package is the Python support for the core scientific dataset (CSD) model file exchange-format[1]. 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.

1.1 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*.

[1] Srivastava D.J., Vosegaard T., Massiot D., Grandinetti P.J. (2020) Core Scientific Dataset Model: A lightweight and portable model and file format for multi-dimensional scientific data. PLOS ONE 15(1): e0225953.

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.

1.1.1 CSDM

Description

The root level object of the CSD model.

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.

1.1.2 Dimension

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

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.

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 kth 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 kth 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 kth dimension, then the coordinates, $\mathbf{X}_k$, and the absolute coordinates, $\mathbf{X}_k^{\text{abs}}$, along a monotonic dimension follow

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

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 kth 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"]
```

1.1.3 DependentVariable

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

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

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.

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.

1.1.4 Enumeration

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.

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.

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

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

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

- $\mathrm{symmetric\_matrix\_n}$ A dependent variable with $p = n^2$ components interpret as a matrix symmetric about its leading diagonal as shown below,

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

- $\mathrm{pixel\_n}$ 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.

1.1.5 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)

1.2 *csdmpy* package

1.2.1 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
```

1.2.2 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)

1.3 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.'
```

1.3.1 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 list object. To access these lists, 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.

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.'
```

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])
```

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`.

1.3.2 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.

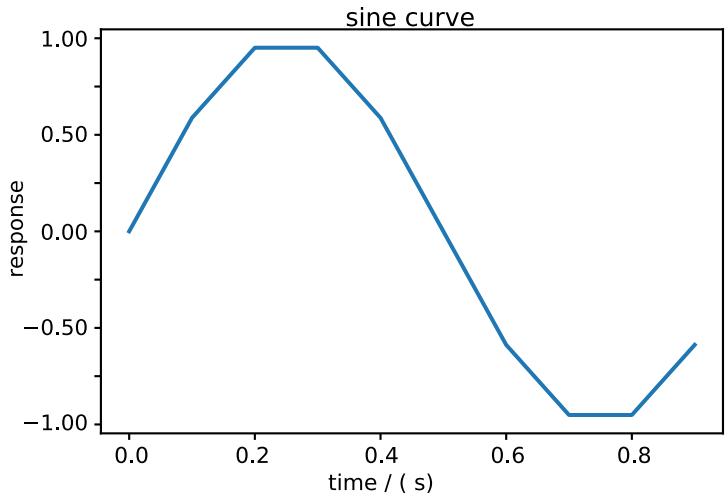
Attention: Although we show code for visualizing the dataset, this documentation is not a guide for data visualization.

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



1.4 Example Gallery

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 $d\mathrm{D}\{p\}$ 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.*, $d\mathrm{D}\{p_0, p_1, p_2, ...\}$.

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

1.4.1 Scalar, 1D{1} datasets

The $1\mathrm{D}\{1\}$ datasets are one dimensional, $d = 1$, with one single-component, $p = 1$, dependent variable. These datasets are the most common, and we, therefore, provide a few examples from various fields of science.

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[1]. The original dataset was downloaded as a CSV file and subsequently converted to the CSD model format.

Let's import this file.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/vetjm3cndxdps05ijvv603ajth3jocck.csdf"
sea_level = cp.load(filename)
```

The variable *filename* is a string with the address to the *.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.

[1] Church JA, White NJ. Sea-Level Rise from the Late 19th to the Early 21st Century. Surveys in Geophysics. 2011;32:585602. DOI:10.1007/s10712-011-9119-1

```
print(sea_level.data_structure)
```

Out:


```
{
  "csdm": {
    "version": "1.0",
    "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 tuple 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)
```

Out:

```
[1880.04166667 1880.125      1880.20833333 ... 2013.79166666 2013.87499999
 2013.95833333] yr
```

and

```
print(y[0].components[0])
```

Out:

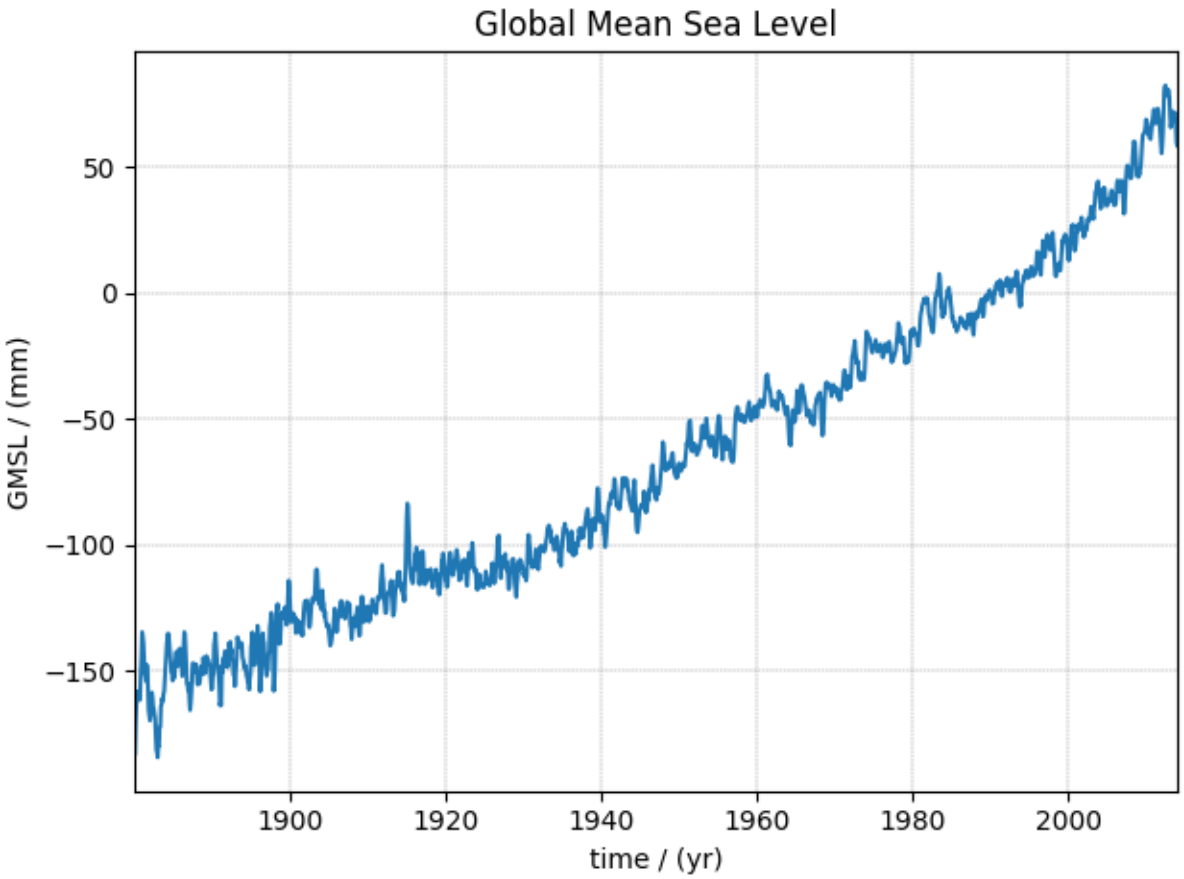
```
[-183.     -171.125  -164.25   ...   66.375    59.6875   58.5   ]
```

respectively.

Plotting the data

Note: The following code is only for illustrative purposes. The users may use any plotting library to visualize their datasets.

```
import matplotlib.pyplot as plt

plt.plot(x[0].coordinates, y[0].components[0].real)
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()
plt.show()
```



The following is a quick description of the above code. Within the code, we make use of the csdm instance's attributes in addition to the matplotlib functions. The first line is an import call for the matplotlib functions. The following line generates a plot of the coordinates along the dimension verse the component of the dependent variable. The next line sets the x-range. For labeling the axes, use the `axis_label` attribute of both dimension and dependent variable instances. For the figure title, use the `name` attribute of the dependent variable instance. The next statement adds the grid lines. For additional information, refer to Matplotlib documentation.

See also:

Getting started with csdmpy package

Citation

Total running time of the script: (0 minutes 1.814 seconds)

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 steps described in the previous example.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/2e4fqm8n8bh4i5wgrinbwcavafa8x7y1.csdf"
NMR_data = cp.load(filename)
print(NMR_data.data_structure)
```

Out:


```
{
  "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",
          "label": "13C frequency shift"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "numeric_type": "complex128",
        "quantity_type": "scalar",
        "components": [
          [
```


(continues on next page)

(continued from previous page)

```
            "(-8899.40625-1276.7734375j), (-4606.88037109375-742.4124755859375j), ...,
↪ (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
```

Out:

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

The *tags* attribute is a list of keywords that best describe the dataset. The *tags* attribute is accessed through,

```
NMR_data.tags
```

Out:

```
['13C', 'NMR', 'spectrum', 'ethanol']
```

You may add additional tags, if so desired, using the *append* method of python's list class, for example,

```
NMR_data.tags.append("Bloch decay")
NMR_data.tags
```

Out:

```
['13C', 'NMR', 'spectrum', 'ethanol', 'Bloch decay']
```

The coordinates along the dimension are

```
x = NMR_data.dimensions
x0 = x[0].coordinates
print(x0)
```

Out:

```
[-3.000e-01 -2.000e-01 -1.000e-01 ...  4.090e+02  4.091e+02  4.092e+02] ms
```

Unlike the previous example, the data structure of an NMR measurement is a complex-valued dependent variable. The numeric type of the components from a dependent variable is accessed through the `numeric_type` attribute.

```
y = NMR_data.dependent_variables
print(y[0].numeric_type)
```

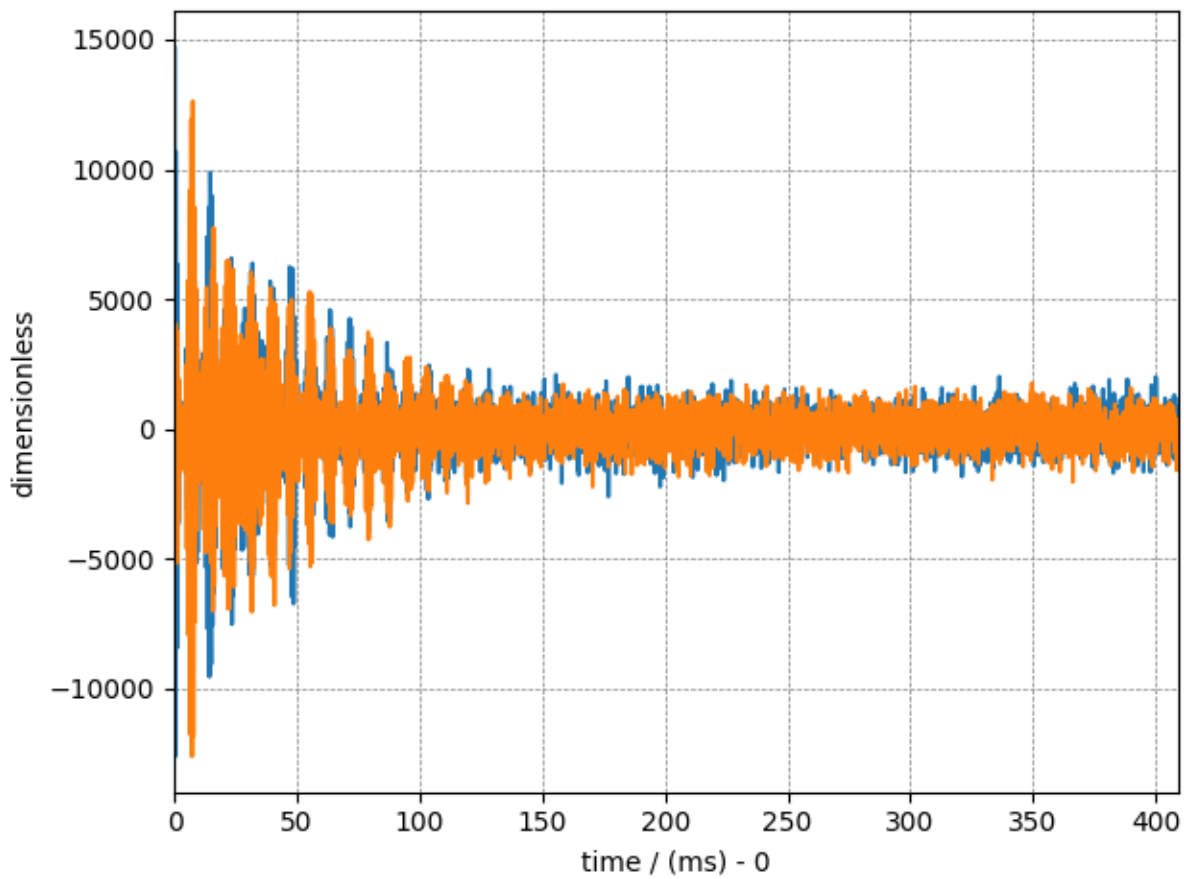
Out:

```
complex128
```

Visualizing the dataset

In the previous example, we illustrated a matplotlib script for plotting 1D data. Here, we use the csdmpy `plot()` method, which is a supplementary method for plotting 1D and 2D datasets only.

```
cp.plot(NMR_data)
```



Reciprocal dimension object

When closely observing the dimension instance of *NMR_data*,

```
print(x[0].data_structure)
```

Out:

```
{
  "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",
```

(continues on next page)

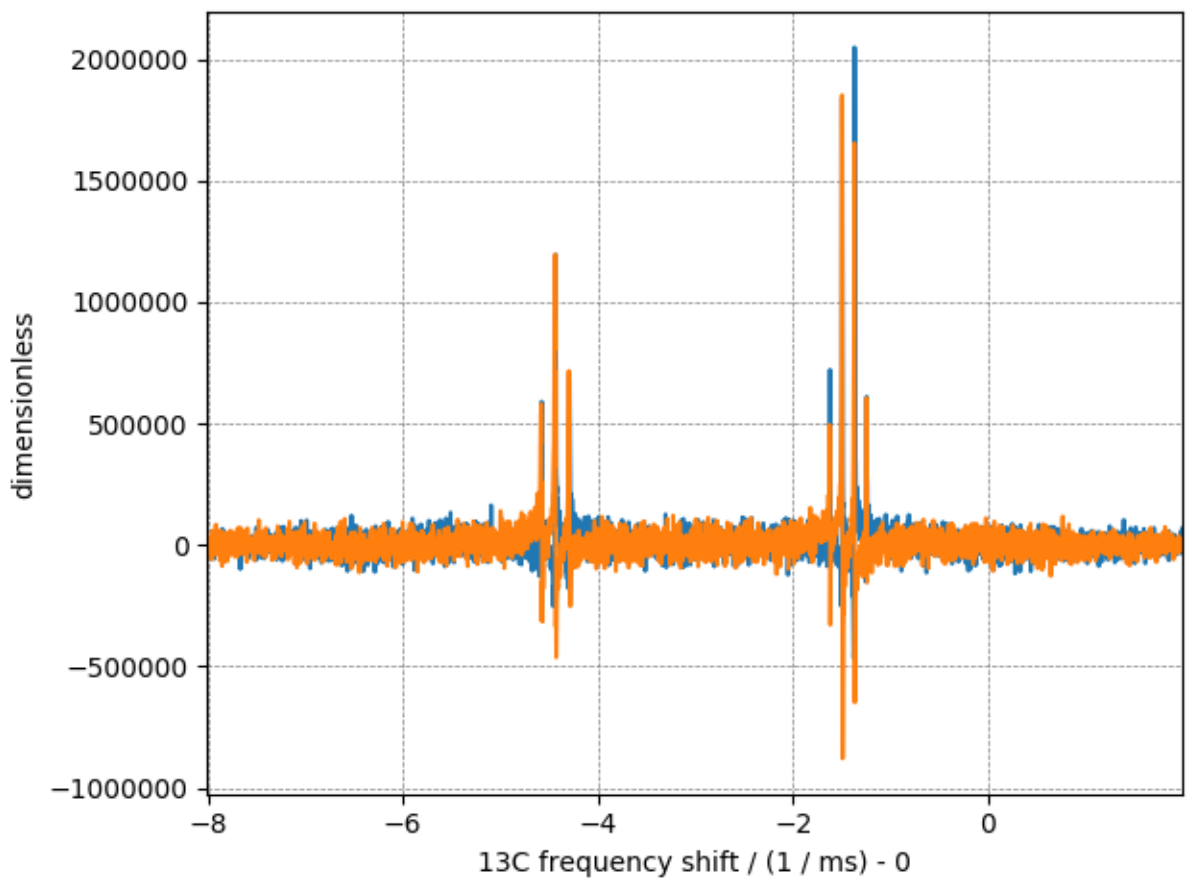
(continued from previous page)

```
    "quantity_name": "frequency",
    "label": "13C frequency shift"
  }
}
```

notice, there is a reciprocal keyword. The `reciprocal` attribute is useful for datasets that frequently transform to a reciprocal domain, such as the NMR dataset. The value of the reciprocal attribute is the reciprocal object, which contains metadata for describing the reciprocal coordinates, such as the *coordinates_offset*, *origin_offset* of the reciprocal dimension.

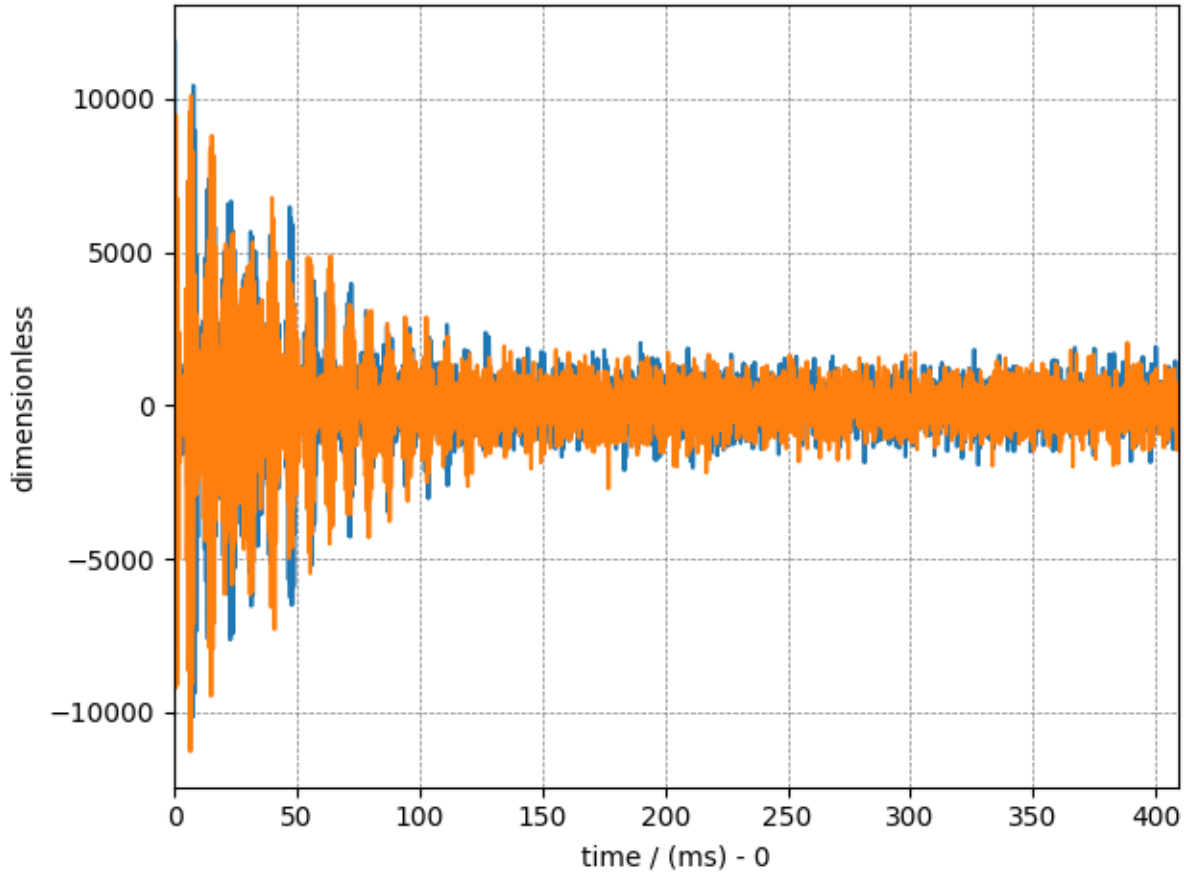
You may perform a fourier transform to visualize the NMR spectrum. Use the `fft()` method on the csdm object `NMR_data` as follows

```
fft_NMR_data = NMR_data.fft()
cp.plot(fft_NMR_data)
```



To return to time domain signal, use the `fft()` method on the `fft_NMR_data` object,

```
NMR_data_2 = fft_NMR_data.fft()
cp.plot(NMR_data_2)
```



Total running time of the script: (0 minutes 1.545 seconds)

Electron Paramagnetic Resonance (EPR) dataset

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

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/0dh8mwnjr600lh1ufpsmt5780yp7wi99.csdf"
EPR_data = cp.load(filename)
print(EPR_data.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2015-02-26T16:41:00Z",
    "description": "A Electron Paramagnetic Resonance simulated dataset.",
    "dimensions": [
      {
        "type": "linear",
```

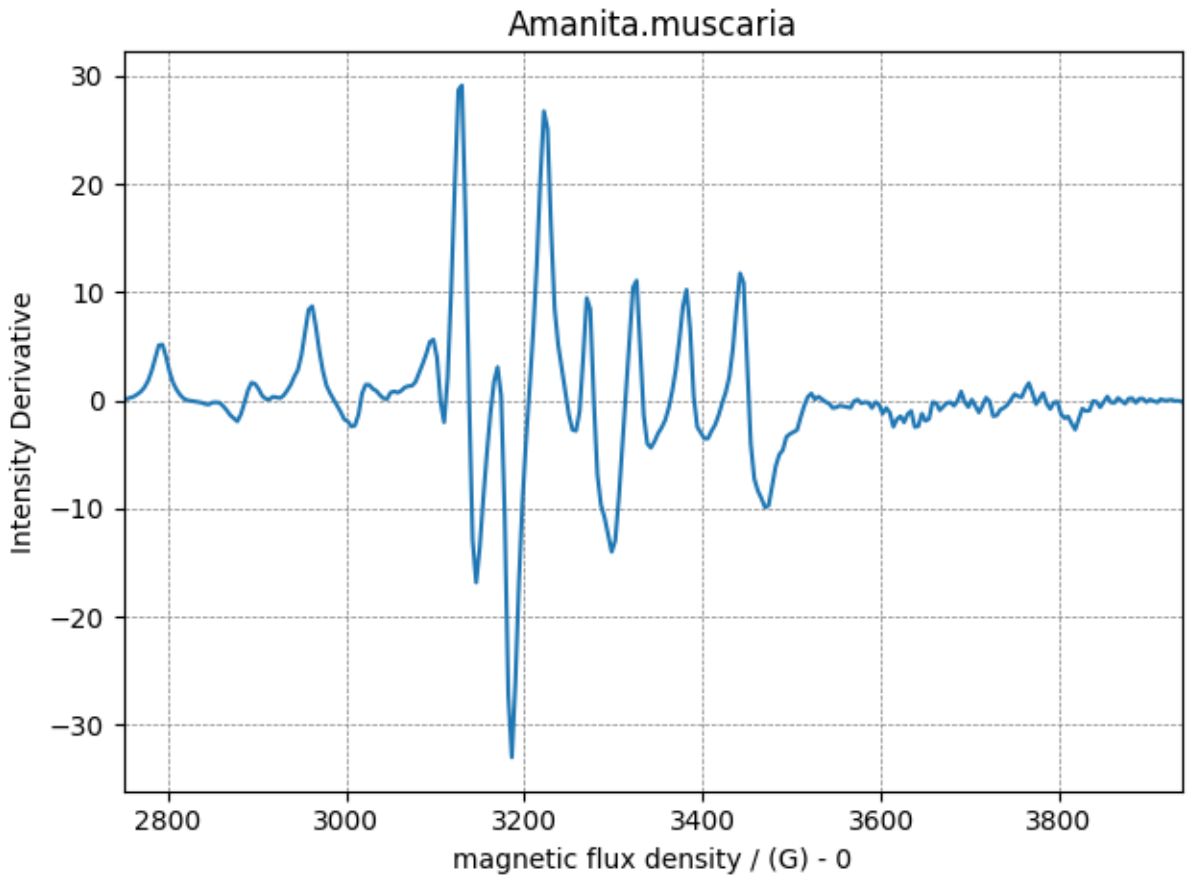
(continues on next page)

(continued from previous page)


```
        "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"
        ],
        "components": [
          [
            "0.067, 0.136, ..., -0.035, -0.137"
          ]
        ]
      }
    ]
  }
}
```


and the corresponding plot

```
cp.plot(EPR_data)
```



Total running time of the script: (0 minutes 1.082 seconds)

Gas Chromatography dataset

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

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/zt452x7p3plbnjqt2898dy8px6hkhkd8.csdf"
GCData = cp.load(filename)
print(GCData.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2011-12-16T12:24:10Z",
    "description": "A Gas Chromatography dataset of cinnamon stick.",
    "dimensions": [
      {
        "type": "linear",
```

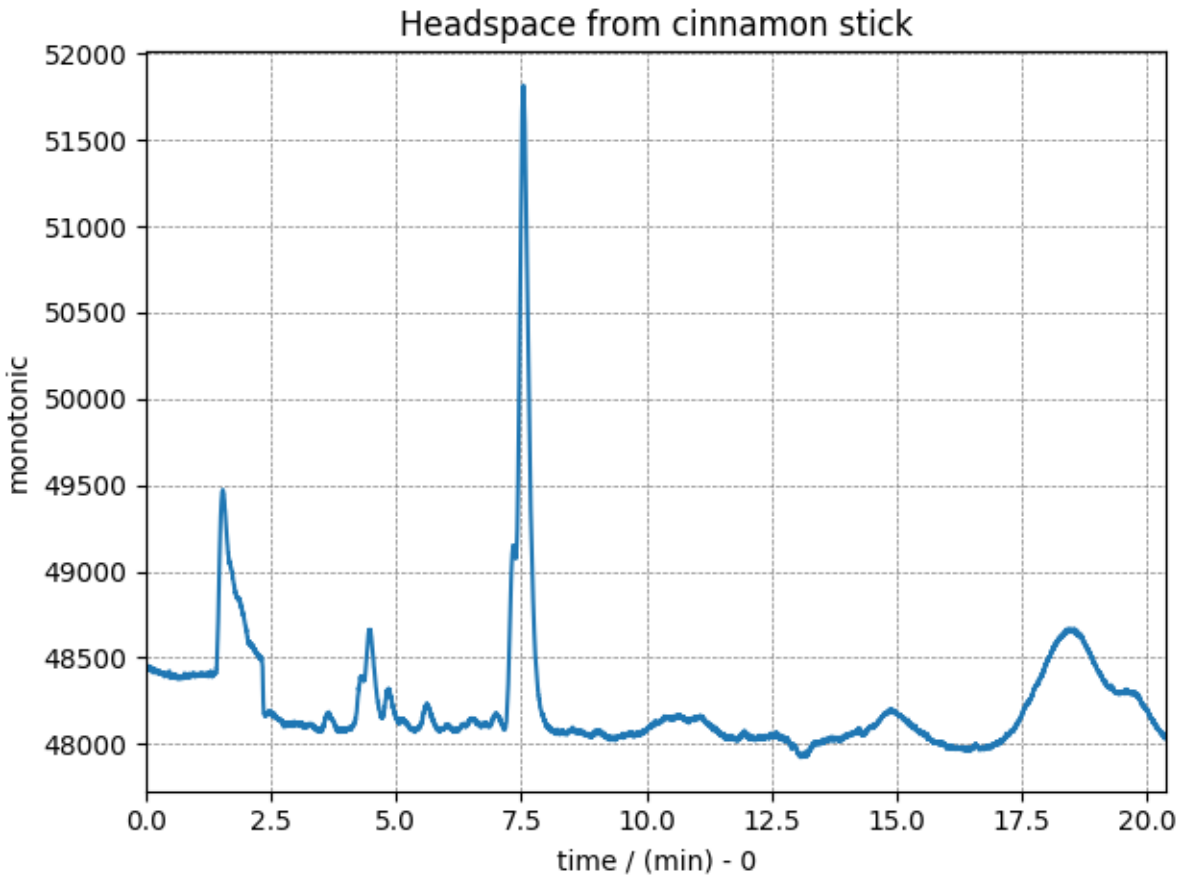
(continues on next page)

(continued from previous page)


```
        "count": 6001,
        "increment": "0.0034 min",
        "quantity_name": "time",
        "reciprocal": {
          "quantity_name": "frequency"
        }
      }
    ],
    "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"
          ]
        ]
      }
    ]
  }
}
```


and the corresponding plot

```
cp.plot(GCData)
```



Total running time of the script: (0 minutes 1.184 seconds)

Fourier Transform Infrared Spectroscopy (FTIR) dataset

The following FTIR dataset, was obtained as a JCAMP-DX file, and subsequently converted to the CSD model file-format. The data structure of the FTIR dataset follows,

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/0iw0egupb1hkulkbdq4hagzzhkbvqjkv.csdf"
FTIR_data = cp.load(filename)
print(FTIR_data.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2019-07-01T21:03:42Z",
    "description": "An IR spectrum of caffeine.",
    "dimensions": [
      {
        "type": "linear",
```

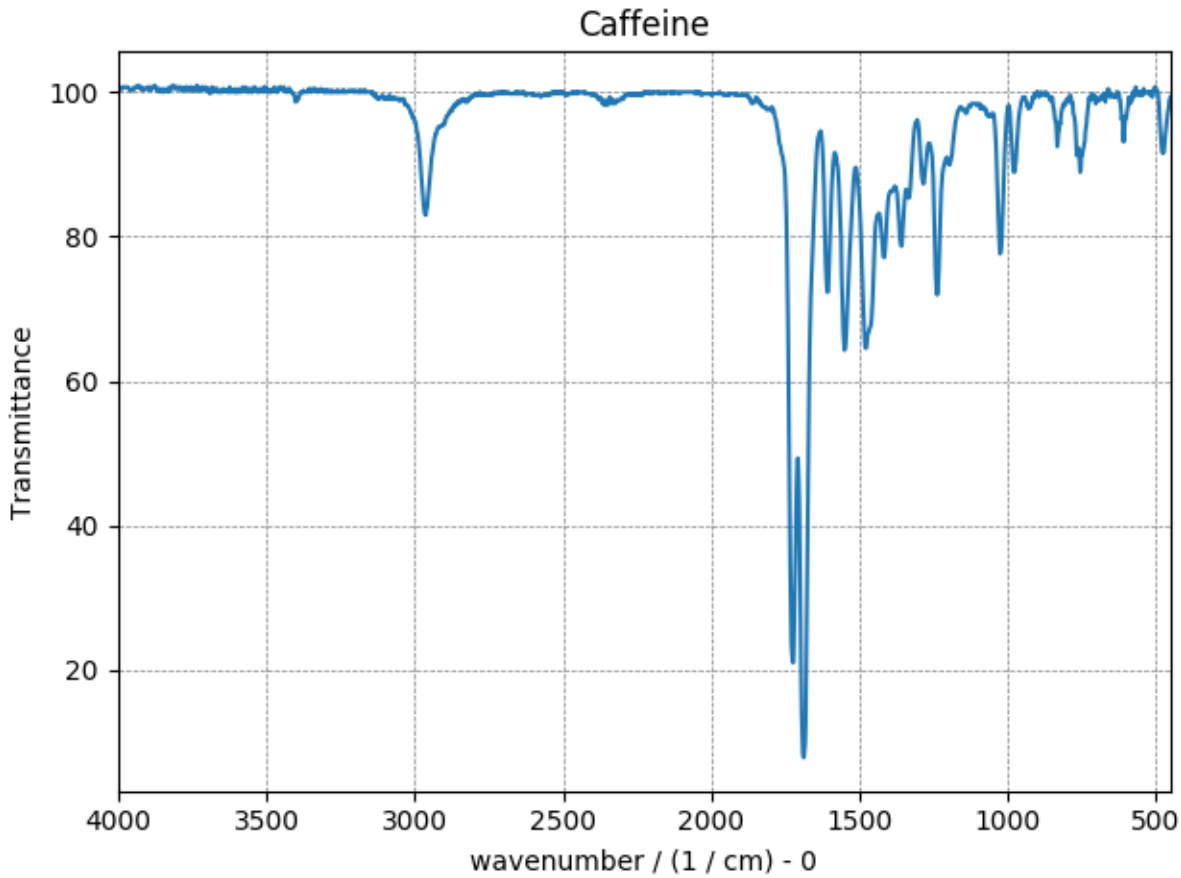
(continues on next page)

(continued from previous page)


```
        "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"
          ]
        ]
      }
    ]
  }
}
```


and the corresponding plot.

```
cp.plot(FTIR_data, reverse_axis=[True])
```



Because, FTIR spectrum is conventionally displayed on a reverse axis, an optional *reverse_axis* argument is provided to the `plot()` method. Its value is an order list of boolean, corresponding to the order of the dimensions.

Total running time of the script: (0 minutes 1.136 seconds)

Ultraviolet–visible (UV-vis) dataset

The following UV-vis dataset was obtained as a JCAMP-DX file, and subsequently converted to the CSD model file-format. The data structure of the UV-vis dataset follows,

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/c9wg59hya5ohc083qi2jgd7wk5emmlmu.csdf"
UV_data = cp.load(filename)
print(UV_data.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2014-09-30T11:16:33Z",
    "description": "A UV-vis spectra of benzene vapours.",
    "dimensions": [
```

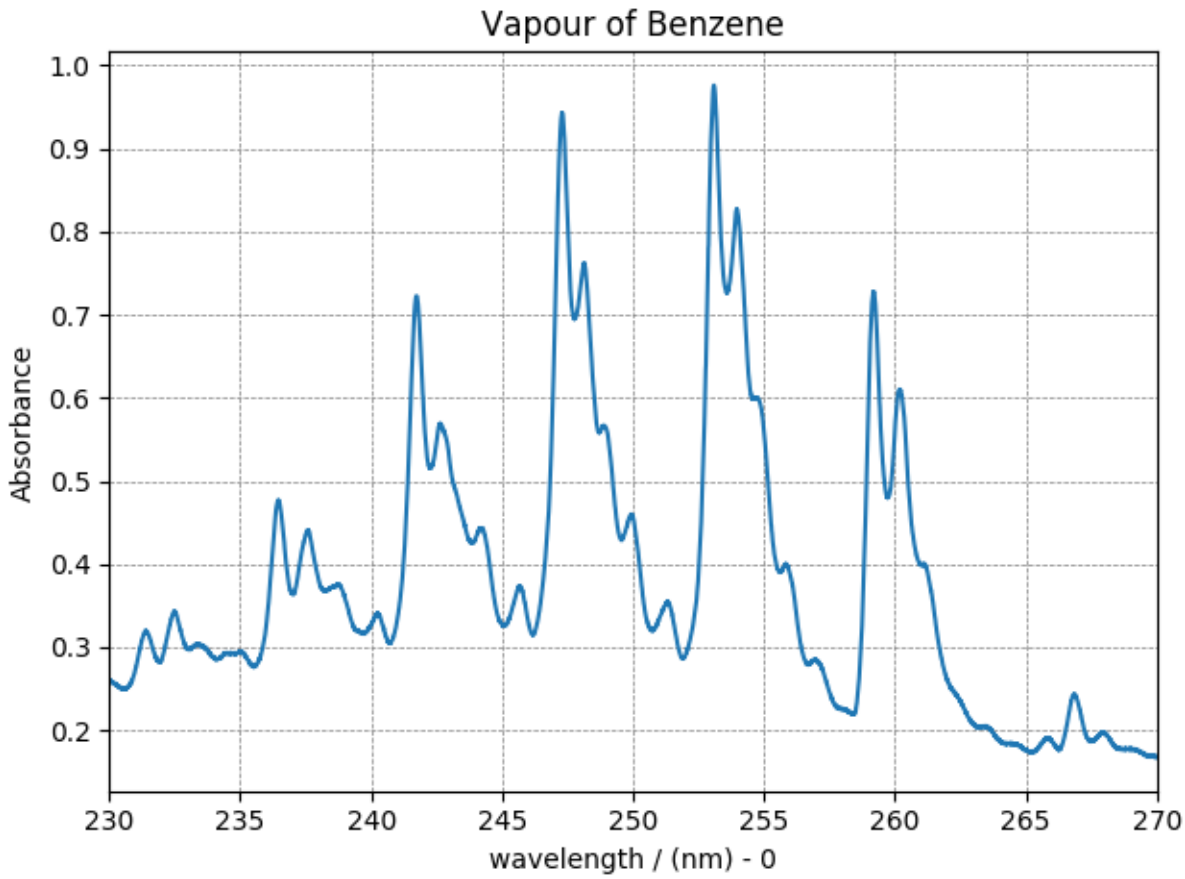
(continues on next page)

(continued from previous page)


```
      {
        "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"
          ]
        ]
      }
    ]
  }
}
```


and the corresponding plot

```
cp.plot(UV_data)
```



Total running time of the script: (0 minutes 1.020 seconds)

Mass spectrometry (sparse) dataset

The following mass spectrometry data of acetone is an example of a sparse dataset. Here, the CSDM data file holds a sparse dependent variable. Upon import, the components of the dependent variable sparsely populates the coordinate grid. The remaining unpopulated coordinates are assigned a zero value.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/ul3rajps49zfuz9ozj3j5xsjmgeuybuy.csdf"
mass_spec = cp.load(filename)
print(mass_spec.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2019-06-23T17:53:26Z",
    "description": "MASS spectrum of acetone",
    "dimensions": [
      {
```

(continues on next page)

(continued from previous page)

```
        "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"
          ]
        ]
      }
    ]
  }
}
```

Here, the coordinates along the dimension are

```
print(mass_spec.dimensions[0].coordinates)
```

Out:

```
[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 corresponding components of the dependent variable,

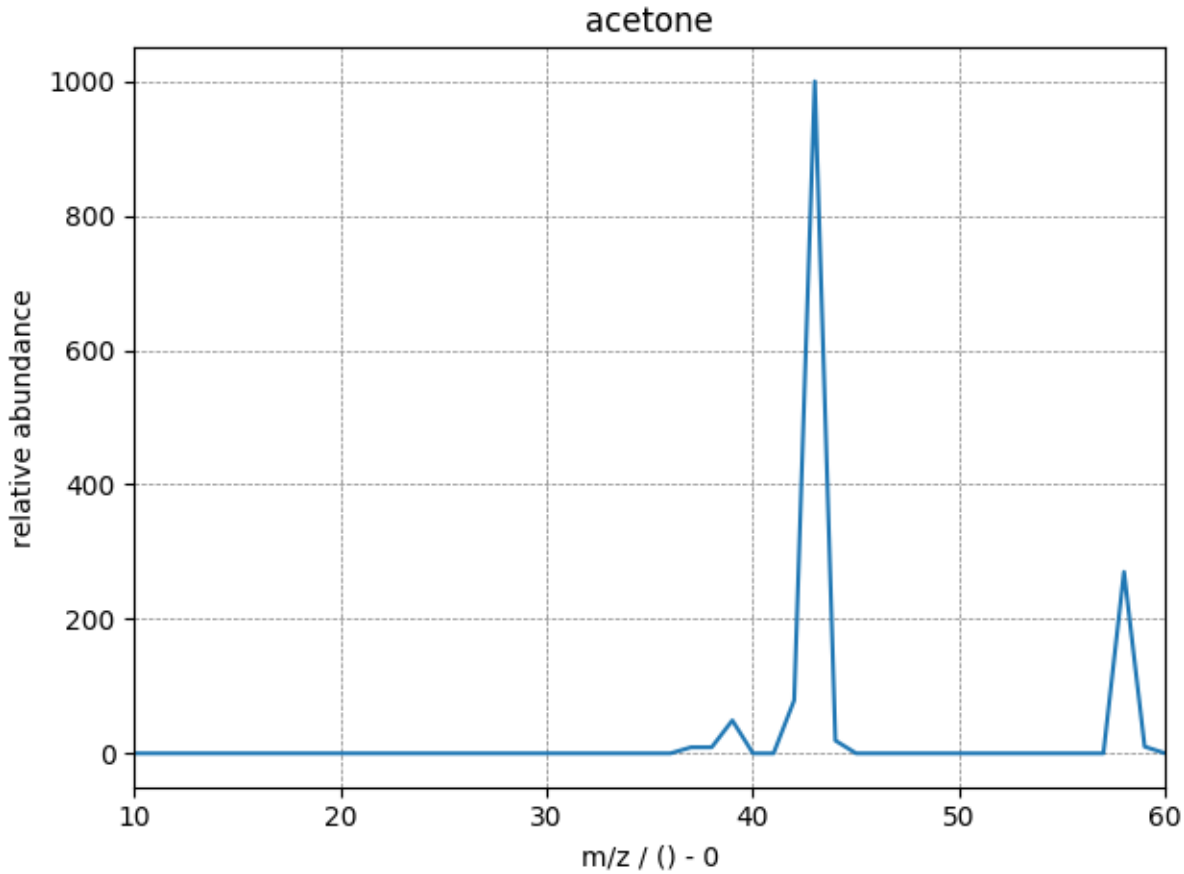
```
print(mass_spec.dependent_variables[0].components[0])
```

Out:

```
[   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 listed in the dependent variable's *components* attribute in the *.csdf* file. The remaining component values were set to zero.

```
cp.plot(mass_spec)
```



Total running time of the script: (0 minutes 1.133 seconds)

1.4.2 Scalar, 2D{1} datasets

The 2D{1} datasets are two dimensional, $d = 2$, with one single-component dependent variable, $p = 1$. Following are some 2D{1} example datasets from various scientific fields expressed in CSDM format.

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

filename = "https://osu.box.com/shared/static/0p3o1ga1kqno4dk4sooi1rbk29pbs3mm.csdf"
bubble_nebula = cp.load(filename)
print(bubble_nebula.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "timestamp": "2020-01-04T01:43:31Z",
    "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 tuple 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])
```

Out:

```
[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])
```

Out:

```
[61.12851495 61.12863695 61.12875894 61.12888094 61.12900293 61.12912493
 61.12924692 61.12936892 61.12949092 61.12961291] deg
```

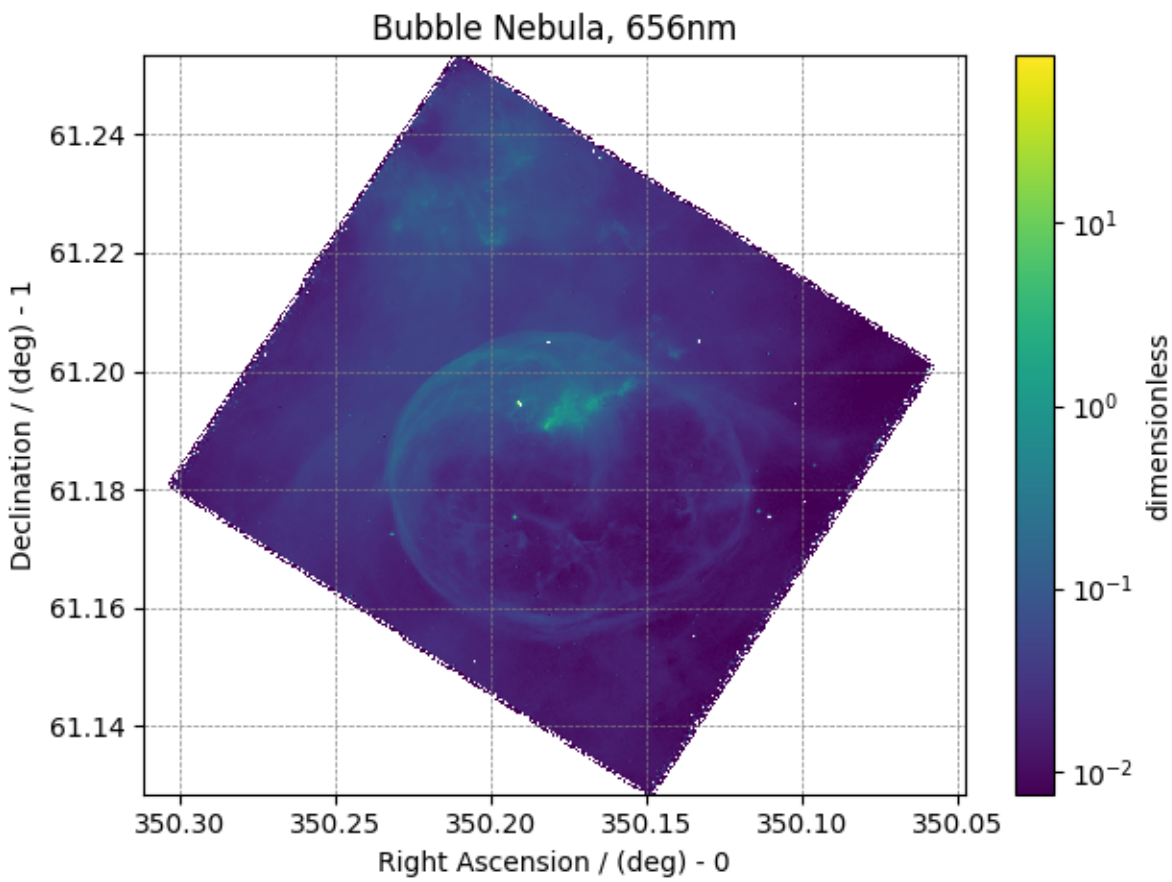
Here, we only print the first ten coordinates along the respective dimensions.

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

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

Visualize the dataset

```
from matplotlib.colors import LogNorm

cp.plot(bubble_nebula, norm=LogNorm(vmin=7.5e-3, clip=True))
```



Note: For 2D{1} datasets, the `plot()` method utilizes the matplotlib *imshow* method to render figures. Any additional arguments provided to the `plot()` method becomes the arguments for the matplotlib *imshow* method. In the above example, the argument *norm* is the argument for the *imshow* method.

Total running time of the script: (0 minutes 1.858 seconds)

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 the 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 = "https://osu.box.com/shared/static/27yrgdaubtb4wqj5adbavp2u16c2h7k8.csdf"
NMR_2D_data = cp.load(filename)
print(NMR_2D_data.description)
```

Out:

```
A 29Si NMR magnetization saturation recovery measurement of highly siliceous zeolite␣
↪ZSM-12.
```

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)
```

Out:

```
{
  "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)
```

Out:

```
{
  "type": "monotonic",
  "coordinates": [
    "1 s",
```

(continues on next page)

(continued from previous page)

```
    "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)
```

Out:

```
[-41040. -40960. -40880. ...  40640.  40720.  40800.] us
```

```
x1 = x[1].coordinates
print(x1)
```

Out:

```
[ 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)
```

Out:

```
[-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]
```

Visualize the dataset

The `plot()` method is a very basic supplementary function for quick visualization of 1D and 2D datasets. You may use this function to plot the data from this example, however, we use the following script to visualize the data with projections onto the respective dimensions.

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

# Set the extents of the image.
```

(continues on next page)

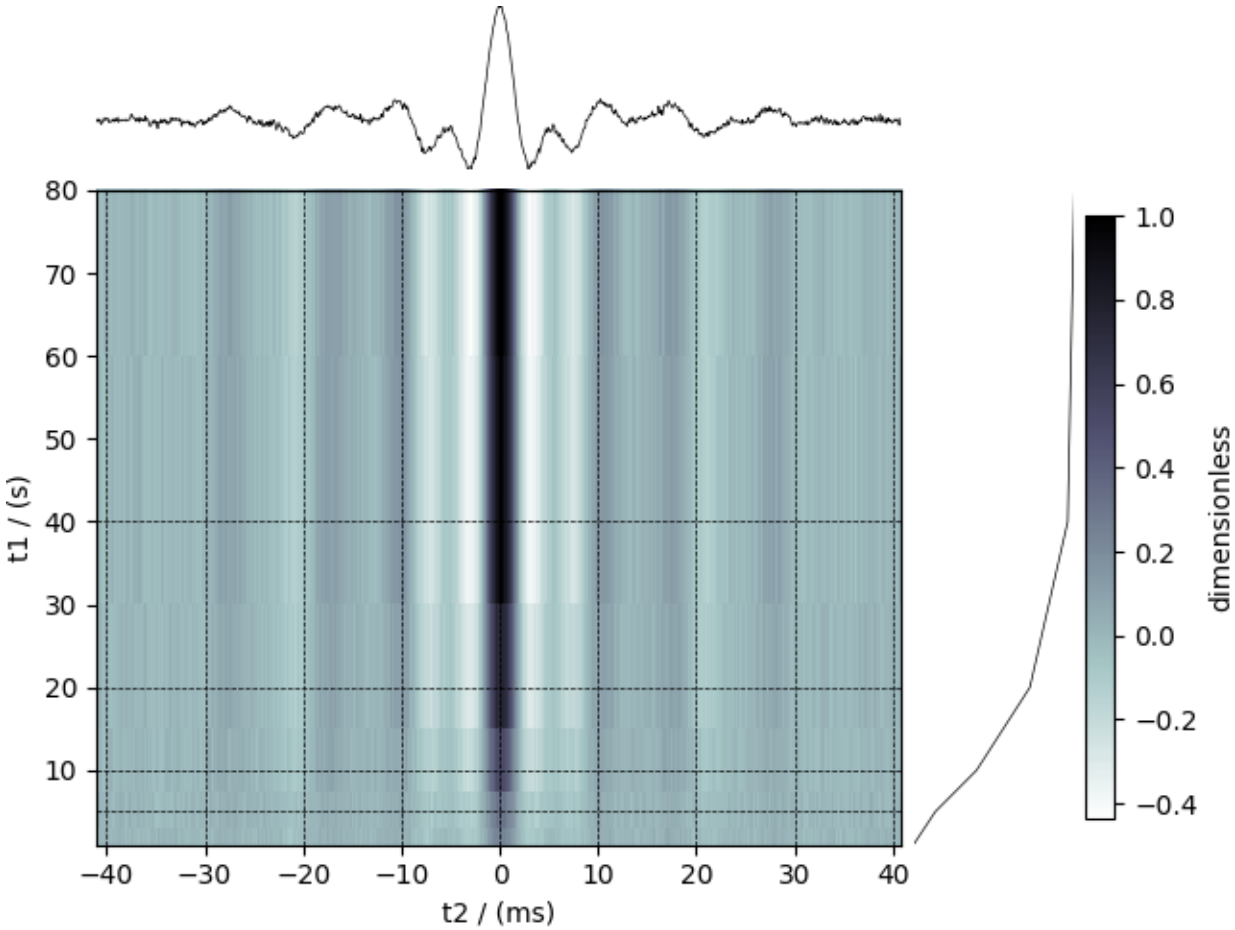
(continued from previous page)

```
# 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.
fig, axi = plt.subplots(
    2, 2, 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)
```

(continues on next page)

(continued from previous page)

```
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=0.0, w_pad=0.0, h_pad=0.0)
plt.subplots_adjust(wspace=0.025, hspace=0.05)
plt.show()
```



Total running time of the script: (0 minutes 1.443 seconds)

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
```

(continues on next page)

(continued from previous page)

```
filename = "https://osu.box.com/shared/static/3w5iqkx15fayan1u6g6sn5woc2ublkyh.csdf"
TEM = cp.load(filename)
print(TEM.data_structure)
```

Out:


```
{
  "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",
        "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 tuple 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])
```

Out:

```
[ 0.  4.  8. 12. 16. 20. 24. 28. 32. 36.] nm
```

```
print(x[1].coordinates[:10])
```

Out:

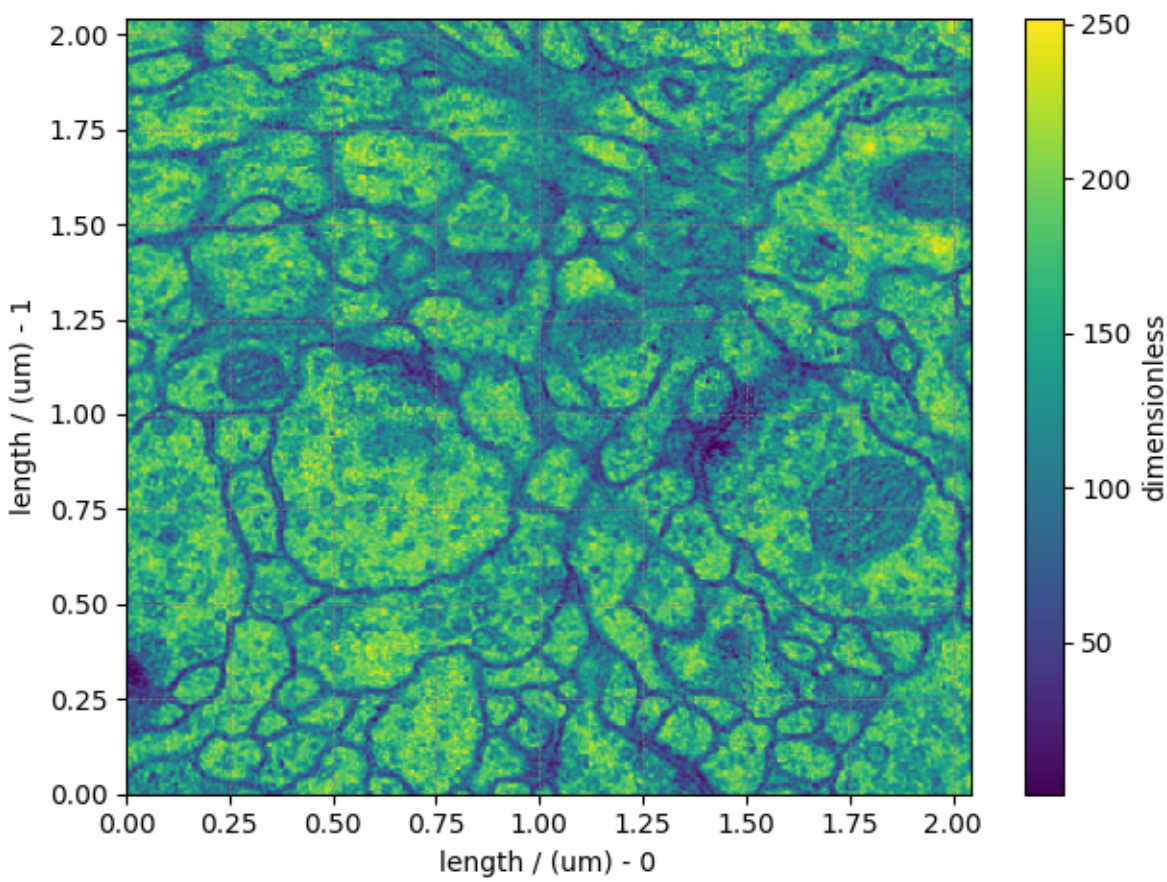
```
[ 0.  4.  8. 12. 16. 20. 24. 28. 32. 36.] nm
```

For convenience, let's convert the coordinates 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.

```
cp.plot(TEM)
```



Total running time of the script: (0 minutes 1.348 seconds)

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 time. The dataset is obtained from The World Bank.

Import the *csdmpy* model and load the dataset.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/e81to3izj5yv5m7mjq9xw7gmqez2blto.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
```

Out:

```
'linear'
```

```
x[1].type
```

Out:

```
'labeled'
```

Here, the second dimension is the *labeled* dimension with[1]

```
x[1].count
```

Out:

```
263
```

labels, where the first five labels are

```
print(x[1].labels[:5])
```

Out:

```
['Aruba' 'Afghanistan' 'Angola' 'Albania' 'Andorra']
```

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

```
print(x[1].coordinates[:5])
```

[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.

Out:

```
['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
```

Out:

```
[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.

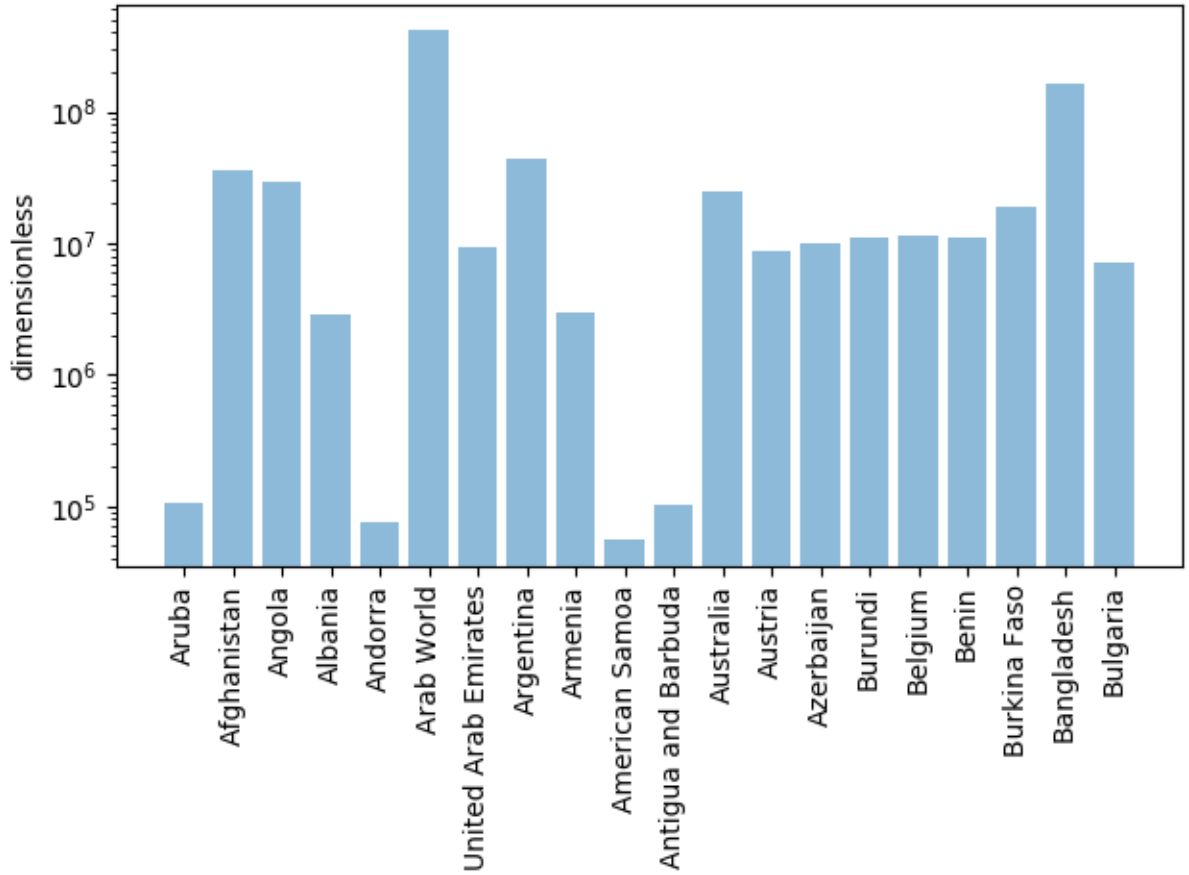
```
print(x[0].coordinates[57])
```

Out:

```
2017.0 yr
```

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

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

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()
plt.show()
```



Total running time of the script: (0 minutes 1.541 seconds)

1.4.3 Vector datasets

Vector, 1D{2} dataset

The 1D{2} datasets are one-dimensional, $d = 1$, with two-component dependent variable, $p = 2$. Such datasets are more common with the weather forecast, such as the wind velocity predicting at a location as a function of time.

The following is an example of a simulated 1D vector field dataset.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/w63851uqruzhz6kx9rx5qgk3or2pot91.csdf"
vector_data = cp.load(filename)
print(vector_data.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
    "timestamp": "2019-02-12T10:00:00Z",
```

(continues on next page)

(continued from previous page)


```
    "dimensions": [
      {
        "type": "linear",
        "count": 10,
        "increment": "1.0 m",
        "quantity_name": "length",
        "reciprocal": {
          "quantity_name": "wavenumber"
        }
      }
    ],
    "dependent_variables": [
      {
        "type": "internal",
        "numeric_type": "float32",
        "quantity_type": "vector_2",
        "components": [
          [
            "0.6907923, 0.31292602, ..., 0.40570852, 0.7005596"
          ],
          [
            "0.5603441, 0.06866818, ..., 0.48200375, 0.15077808"
          ]
        ]
      }
    ]
  }
}
```


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

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

with coordinates

```
print(x[0].coordinates)
```

Out:

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] m
```

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)
```

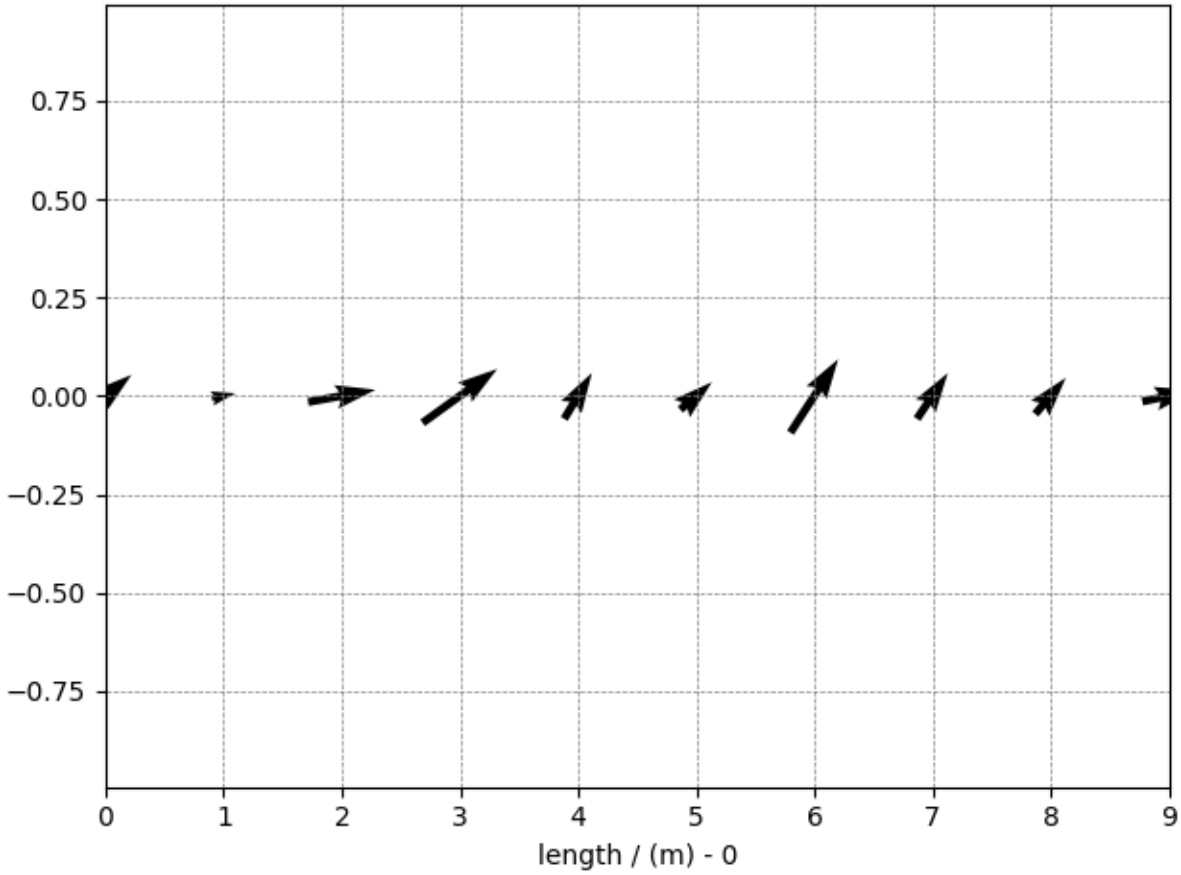
Out:

```
vector_2
```

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

Visualizing the dataset

```
cp.plot(vector_data)
```



Total running time of the script: (0 minutes 1.196 seconds)

Vector, 2D{2} dataset

The 2D{2} datasets are two-dimensional, $d = 2$, with one two-component dependent variable, $p = 2$. 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 = "https://osu.box.com/shared/static/iobasl6fx1z7rds3ovamrwueek8ver5o.csdf"
vector_data = cp.load(filename)
print(vector_data.data_structure)
```

Out:

```
{
  "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",
```

(continues on next page)

(continued from previous page)


```
        "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 tuple 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])
```

Out:

```
[-2.     -1.9375 -1.875  -1.8125 -1.75  ] cm
```

```
print(x[1].coordinates[:5])
```

Out:

```
[-2.     -1.9375 -1.875  -1.8125 -1.75  ] cm
```

The components of the dependent variable are vector components as seen from the `quantity_type` attribute of the corresponding dependent variable instance.

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

Out:

```
vector_2
```

Visualizing the dataset

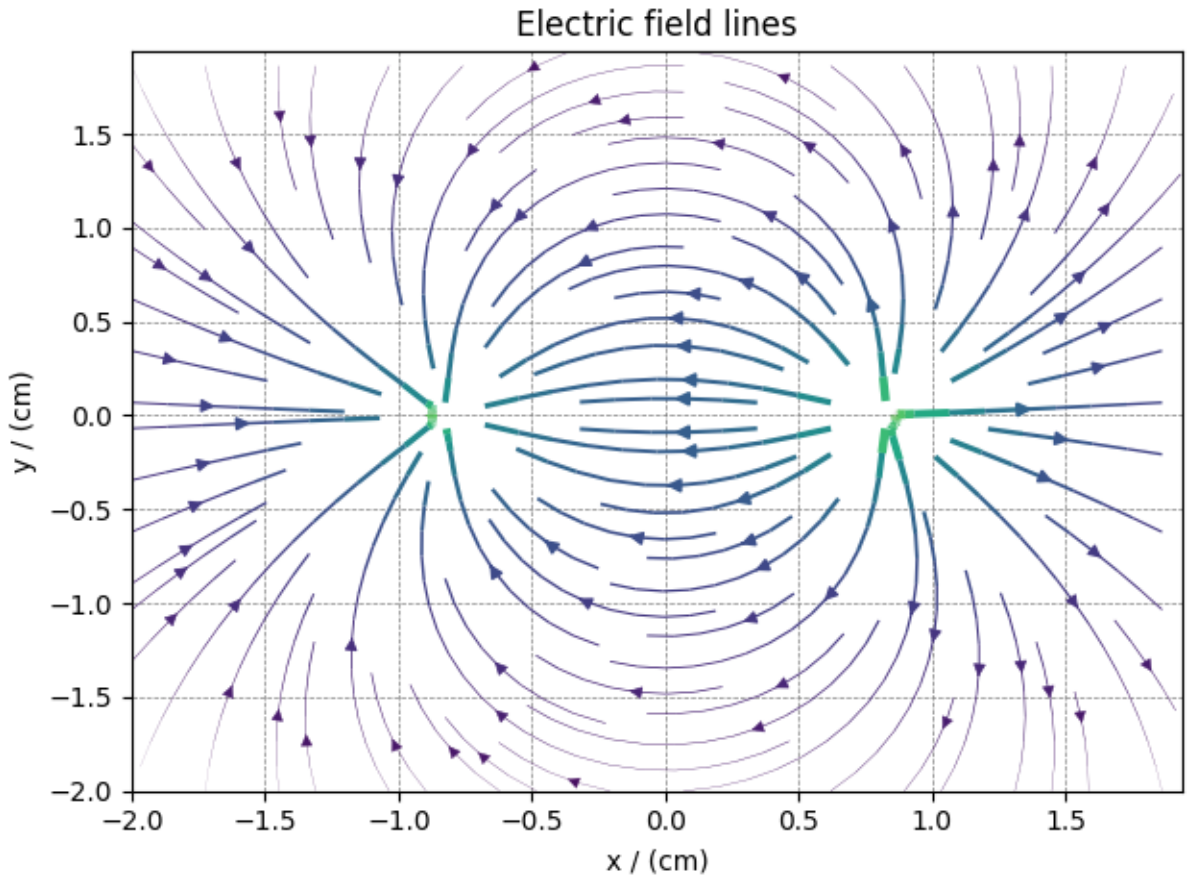
Let's visualize the vector data using the *streamplot* method from the matplotlib package. Before we could visualize, however, there is an initial processing step. We use the Numpy library for processing.

```
import numpy as np

X, Y = np.meshgrid(x[0].coordinates, x[1].coordinates)  # (x, y) coordinate pairs
U, V = y[0].components[0], y[0].components[1]  # U and V are the components
R = np.sqrt(U ** 2 + V ** 2)  # The magnitude of the vector
R /= R.min()  # Scaled magnitude of the vector
Rlog = np.log10(R)  # Scaled magnitude of the vector on a log scale
```

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 streamplot vector plot

```
import matplotlib.pyplot as plt

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()
plt.show()
```



Total running time of the script: (0 minutes 2.312 seconds)

1.4.4 Pixel datasets

Image, 2D{3} datasets

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

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/vdxdaitsa9dq45x8nk7l7h25qrw2baxt.csdf"
ImageData = cp.load(filename)
print(ImageData.data_structure)
```

Out:

```
{
  "csdm": {
    "version": "1.0",
    "read_only": true,
```

(continues on next page)

(continued from previous page)


```
    "timestamp": "2016-03-12T16:41:00Z",
    "tags": [
      "racoon",
      "image",
      "Judy Weggelaar"
    ],
    "description": "An RBG image of a raccoon face.",
    "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 tuple 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])
```

Out:

```
x0 = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

```
print("x1 =", x[1].coordinates[:10])
```

Out:

```
x1 = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

respectively, where only first ten coordinates along each dimension is 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)
```

Out:

```
pixel_3
```

From the value *pixel_3*, *pixel* indicates a pixel data, 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])
```

Out:

```
[[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 the component array is accessed through the `component_labels` attribute with appropriate indexing, that is

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

Out:

```
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)
```

Out:

```
red (768, 1024)
```

```
print(y[0].component_labels[1], y[0].components[1].shape)
```

Out:

```
green (768, 1024)
```

```
print(y[0].component_labels[2], y[0].components[2].shape)
```

Out:

```
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,

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

plt.imshow(np.moveaxis(y[0].components, 0, -1))
plt.xlabel(x[0].axis_label)
plt.ylabel(x[1].axis_label)
plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 1.537 seconds)

1.4.5 Correlated datasets

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*. Following are a few examples of the correlated dataset.

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[1], 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 = "https://osu.box.com/shared/static/h1nxth6gs94fthfmvip5vchp3zh4zd6o.csdf"
zero_d_dataset = cp.load(filename)
```

[1] Srivastava DJ, Florian P, Baltisberger JH, Grandinetti PJ. Correlating geminal couplings to structure in framework silicates. Phys Chem Chem Phys. 2018;20:562571. DOI:10.1039/C7CP06486A

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)
```

Out:

```
[]
```

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)
```

Out:


```
{
  "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)
```

Out:

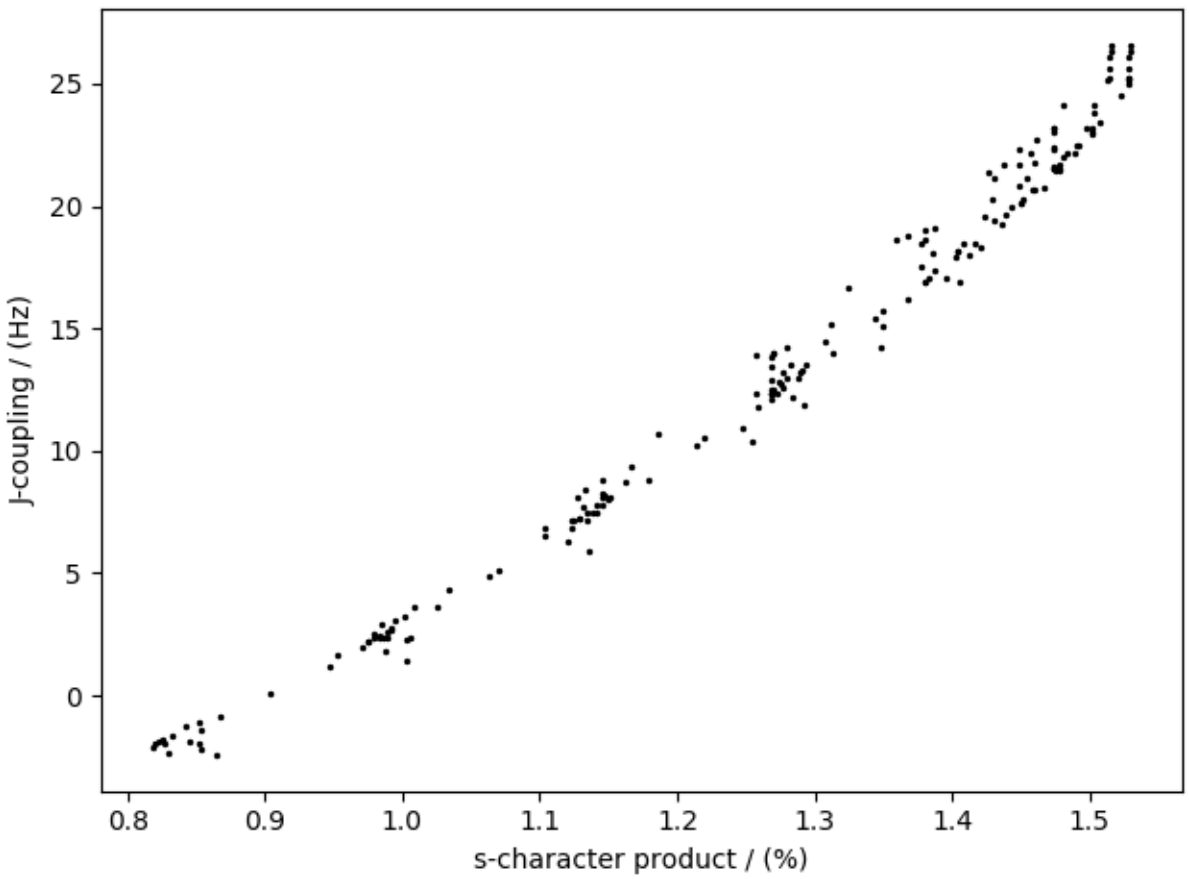

```
{
  "type": "internal",
  "name": "product of s-characters",
  "unit": "%",
  "numeric_type": "float32",
  "quantity_type": "scalar",
  "component_labels": [
    "s-character product"
  ],
  "components": [
    [
      "0.8457453, 0.8534185, ..., 1.5277092, 1.5289451"
    ]
  ]
}
```


respectively.

Visualizing the dataset

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

```
import matplotlib.pyplot as plt

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()
plt.show()
```



Citation

Total running time of the script: (0 minutes 1.180 seconds)

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 variables, 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 = "https://osu.box.com/shared/static/6uhrtdxfisl4a14x9pndyze2mv414zyg.csdf"
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 labels along thee respective dimensions are

```
x[0].label
```

Out:

```
'longitude'
```

```
x[1].label
```

Out:

```
'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)
```

Out:

```
{
  "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"
  ],
```

(continues on next page)

(continued from previous page)

```
  "components": [
    [
      "292.8152160644531, 293.0152282714844, ..., 301.8152160644531, 303.8152160644531
↪"
    ]
  ]
}
```

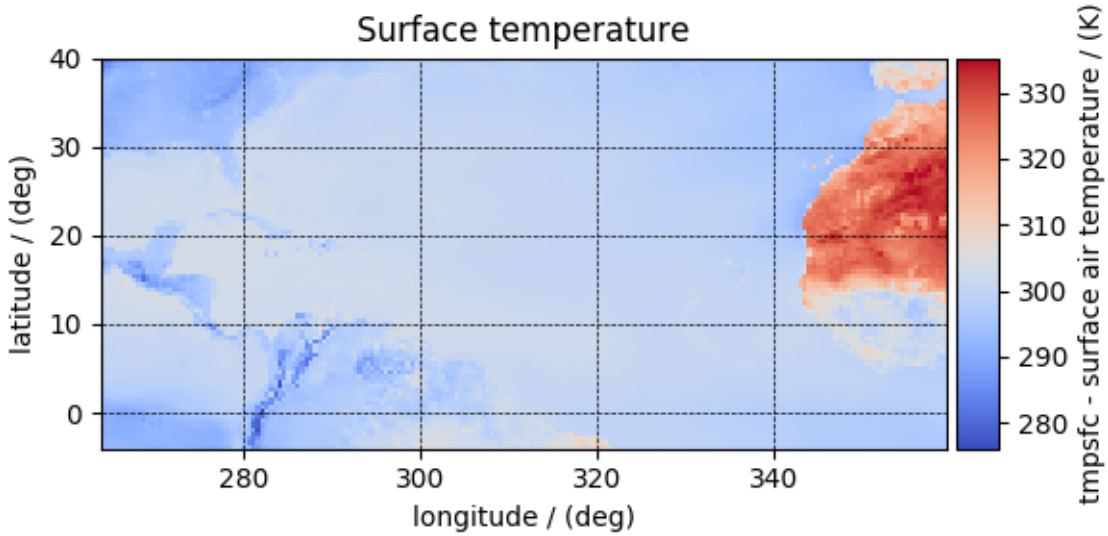
If you have followed all previous examples, the above data structure should be self-explanatory.

We will use the following snippet to plot the dependent variables of scalar *quantity_type*.

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

Now to plot the data from the dependent variable.

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



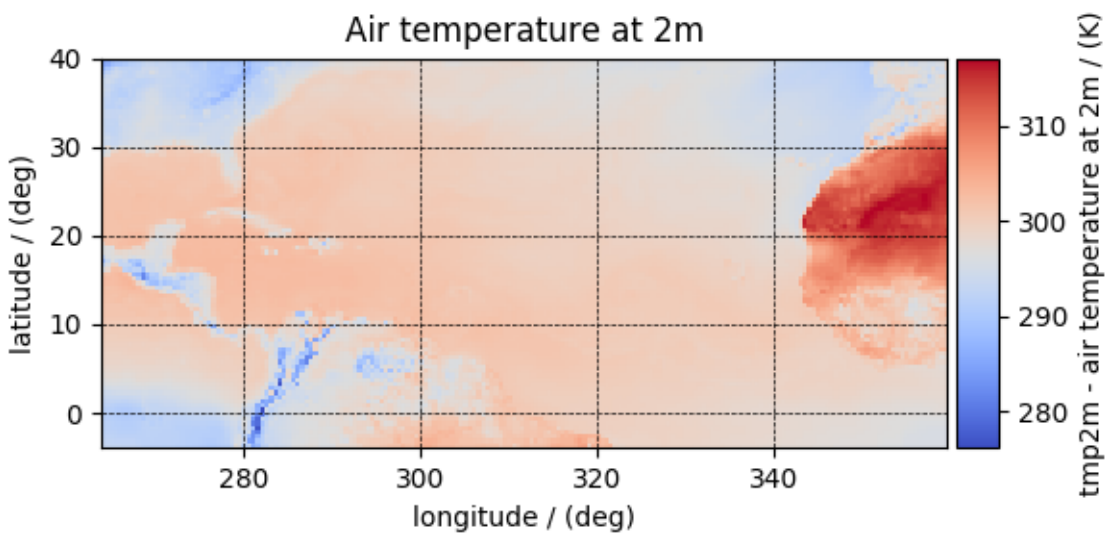
Similarly, other dependent variables with their respective plots are

```
y[1].name
```

Out:

```
'Air temperature at 2m'
```

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

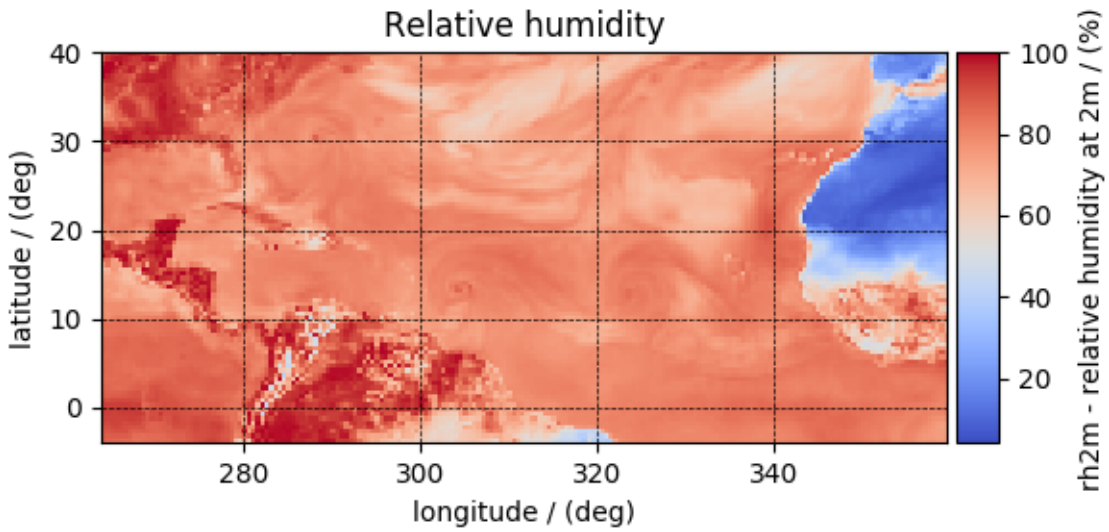



```
y[3].name
```

Out:

```
'Relative humidity'
```

```
plot_scalar(y[3])
```

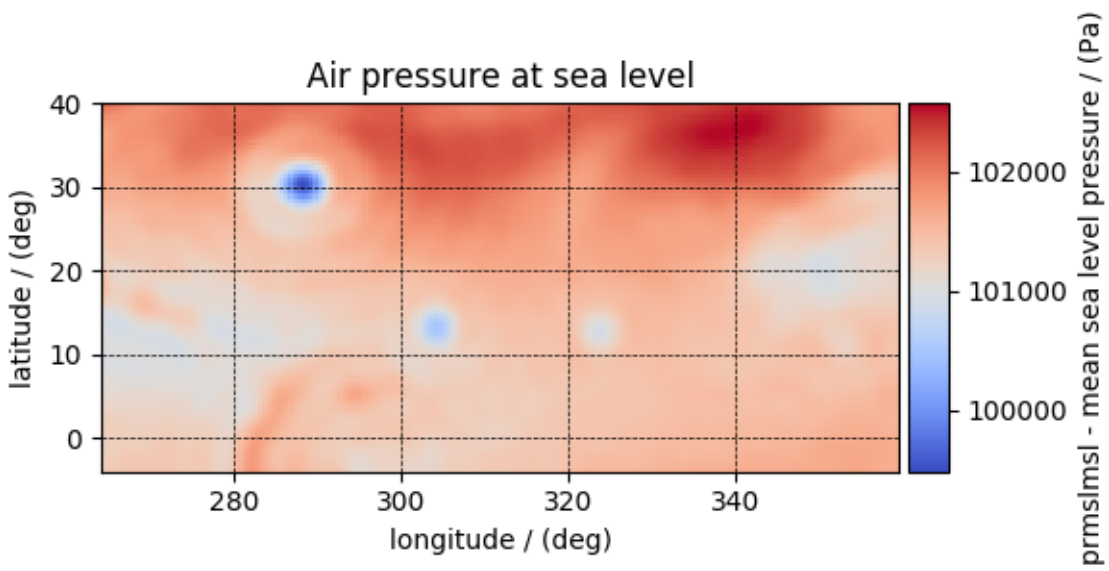



```
y[4].name
```

Out:

```
'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
```

Out:

```
'vector_2'
```

```
y[2].name
```

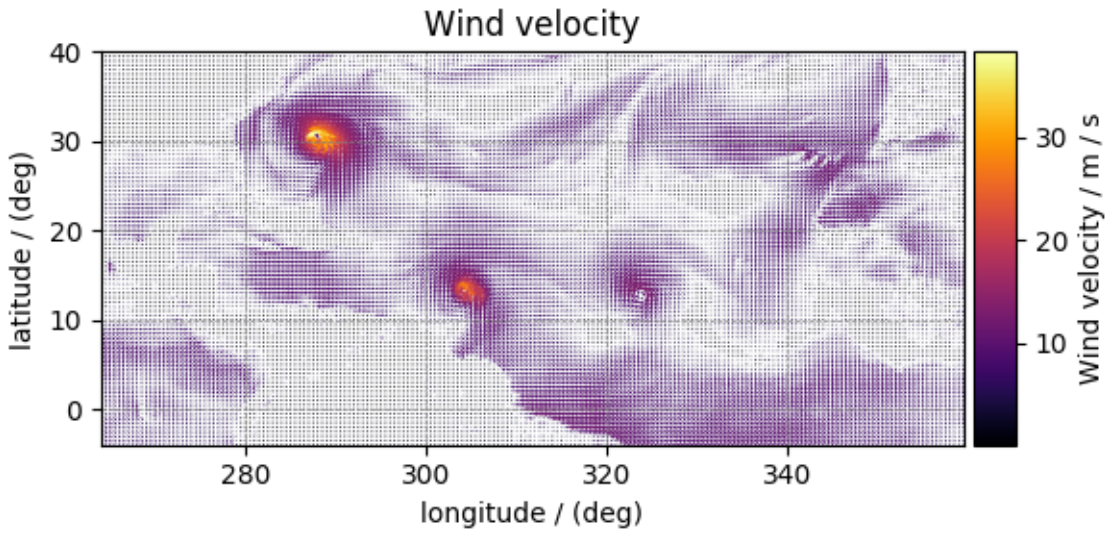
Out:

```
'Wind velocity'
```

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

```
def plot_vector(yx):
    fig, ax = plt.subplots(1, 1, figsize=(6, 3))
    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()
    plt.show()
```

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



Total running time of the script: (0 minutes 2.737 seconds)

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

filename = "https://osu.box.com/shared/static/of3wmoxcqungkp6ndbplnbxtgu6jaahh.csdf"
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
```

Out:

```
'Eagle Nebula acquired @ 502 nm'
```

```
y[1].name
```

Out:

```
'Eagle Nebula acquired @ 656 nm'
```

```
y[2].name
```

Out:

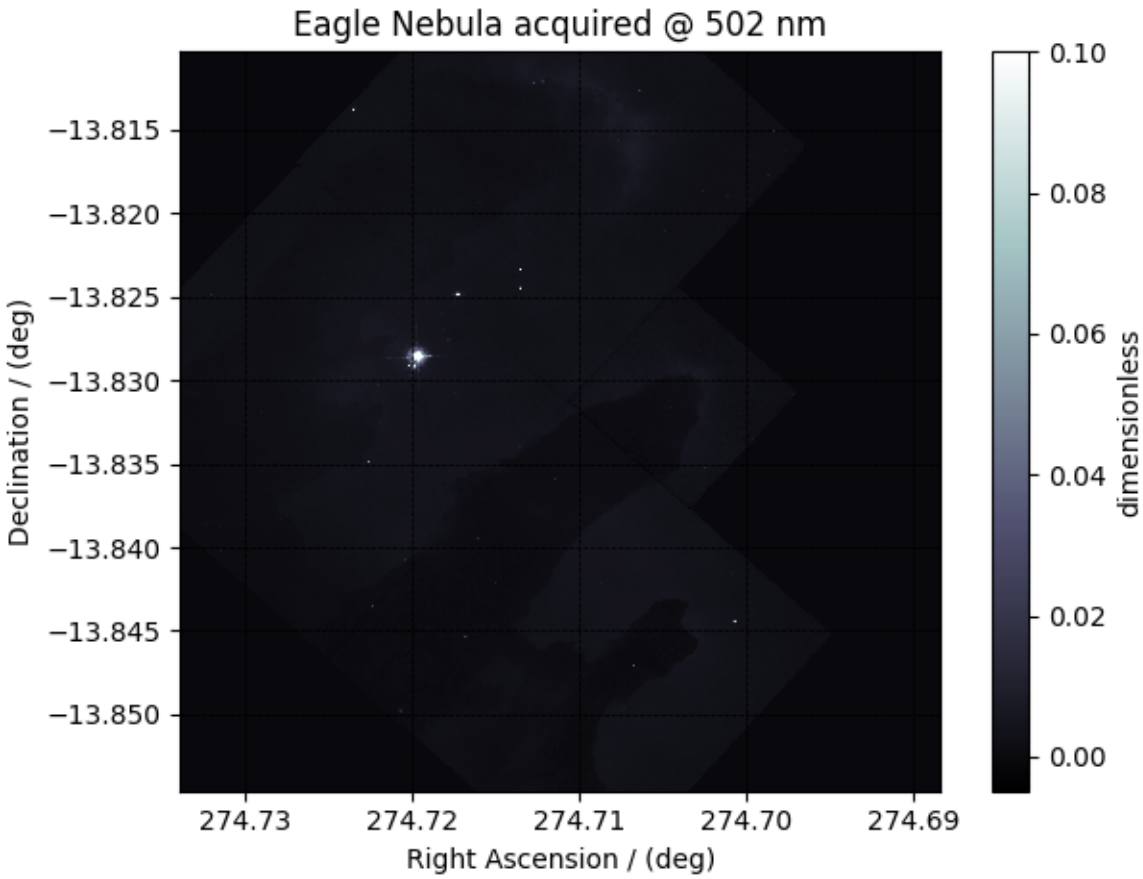
```
'Eagle Nebula acquired @ 673 nm'
```

We use the following script to plot the dependent variables.

```
import matplotlib.pyplot as plt


def plot_scalar(yx):

    # 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.
    cbar = plt.gca().figure.colorbar(im)
    cbar.ax.set_ylabel(yx.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(yx.name)

    # Set up the grid lines.
    plt.grid(color="k", linestyle="--", linewidth=0.5)

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

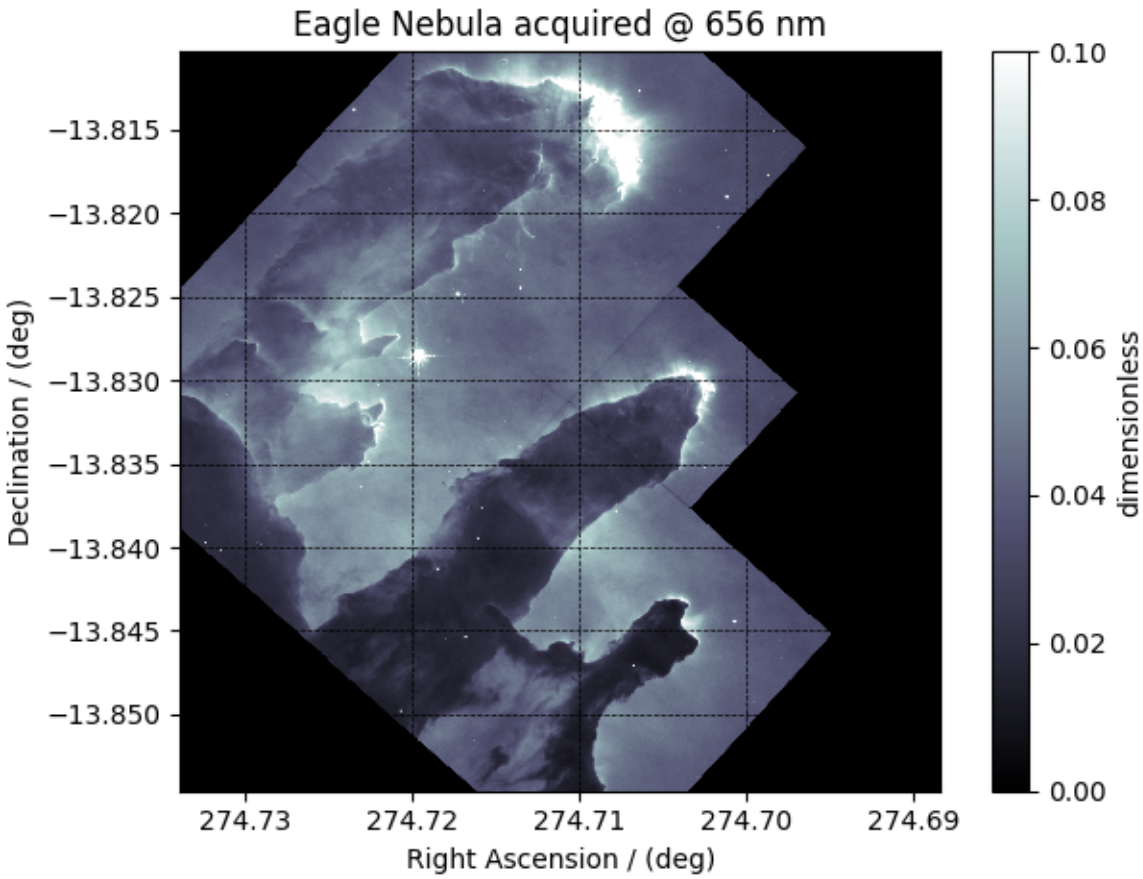
Let's plot the dependent variables, first dependent variable,

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



second dependent variable, and

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



the third dependent variable.

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

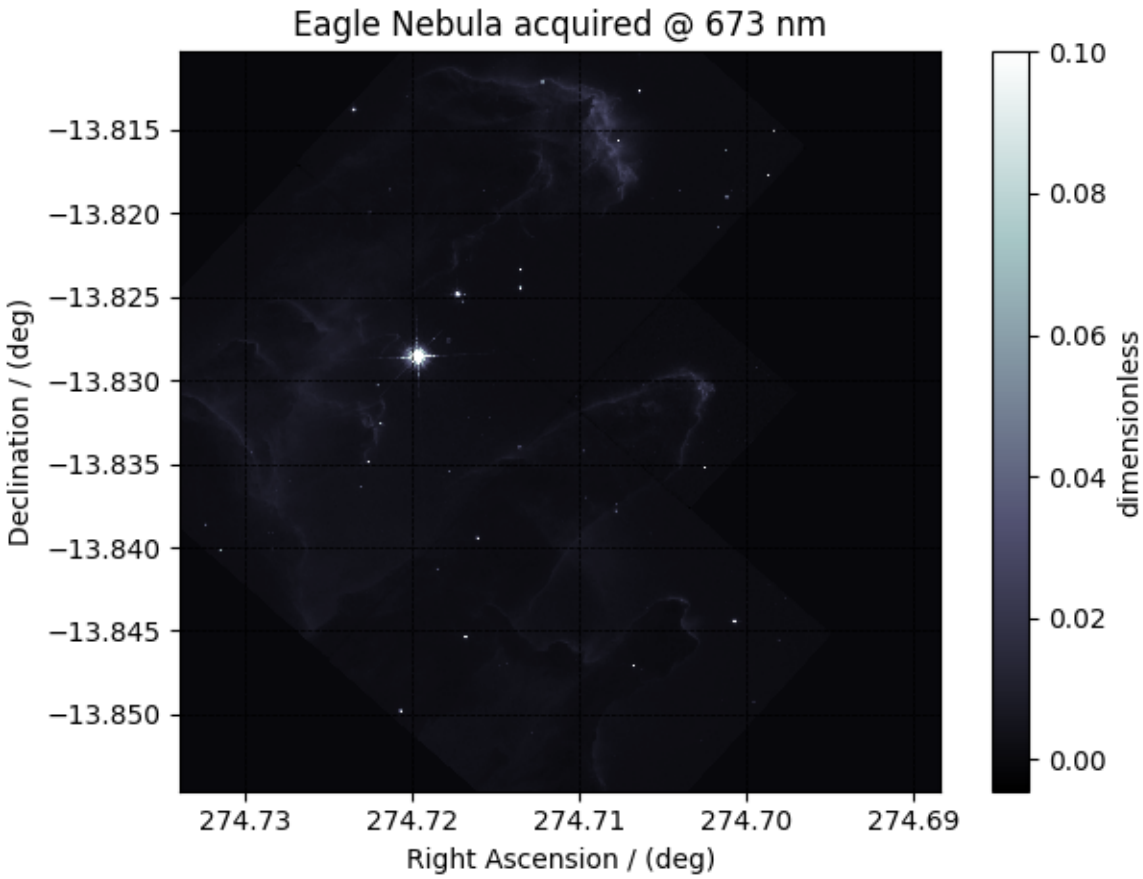


Image composition

```
import numpy as np
```

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,

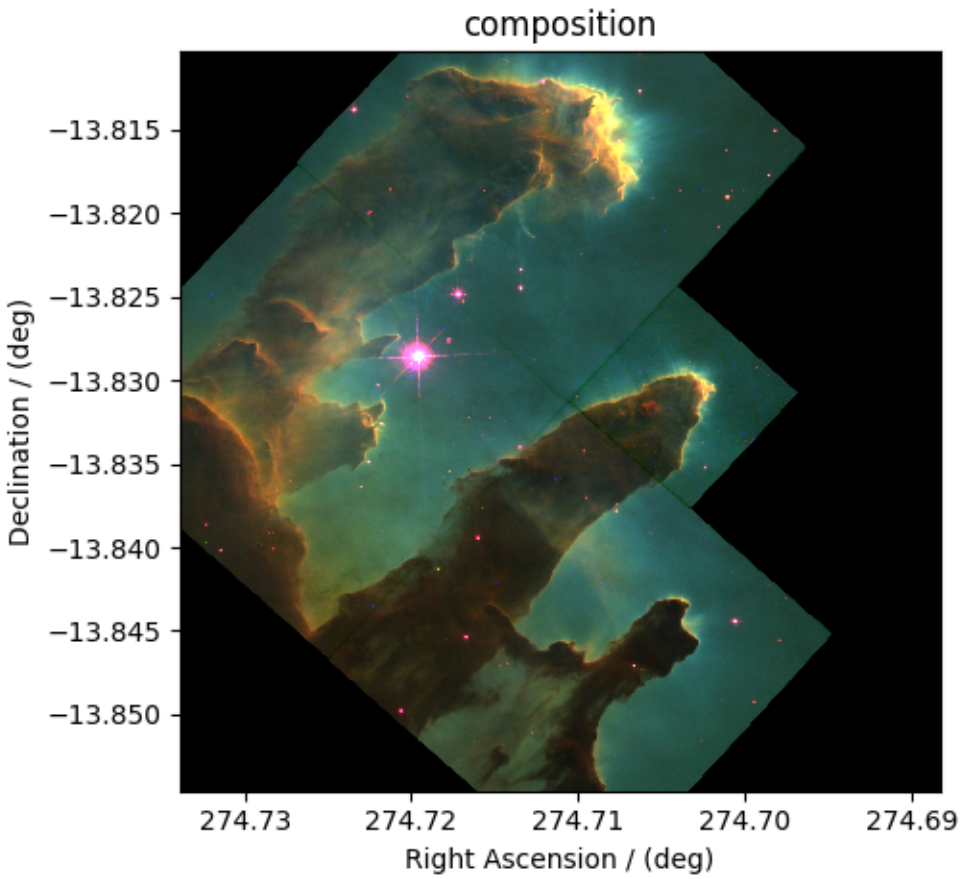
```
image[..., 0] = y[2].components[0] / y[2].components[0].max()   # red channel
image[..., 1] = y[1].components[0] / y[1].components[0].max()   # green channel
image[..., 2] = y[0].components[0] / y[0].components[0].max()   # blue channel
```

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] = np.clip(image[..., 0] * 65.0, 0, 1)  # red channel
image[..., 1] = np.clip(image[..., 1] * 7.50, 0, 1)  # green channel
image[..., 2] = np.clip(image[..., 2] * 75.0, 0, 1)  # blue channel
```

Now to plot this 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.imshow(image, origin="lower", extent=extent)

plt.xlabel(x[0].axis_label)
plt.ylabel(x[1].axis_label)
plt.title("composition")

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



Total running time of the script: (0 minutes 5.580 seconds)

1.4.6 Sparse datasets

Sparse along one dimension, 2D{1,1} dataset

The following is an example[1] of a 2D{1,1} sparse dataset with two-dimensions, $d = 2$, and two, $p = 2$, sparse single-component dependent-variables, where the component is sparsely sampled along one dimension.

Let's import the CSD model data-file.

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/1ltzzbdyeo5bn7xuxmdkxj3e4o1xvvjc.csdf"
sparse_1d = cp.load(filename)
```

There are two linear dimensions and two single-component sparse dependent variables. The tuple of the dimension and the dependent variable instances are

```
x = sparse_1d.dimensions
y = sparse_1d.dependent_variables
```

The coordinates, viewed only for the first ten coordinates, are

```
print(x[0].coordinates[:10])
```

Out:

```
[   0.  192.  384.  576.  768.  960. 1152. 1344. 1536. 1728.] us
```

```
print(x[1].coordinates[:10])
```

Out:

```
[   0.  192.  384.  576.  768.  960. 1152. 1344. 1536. 1728.] us
```

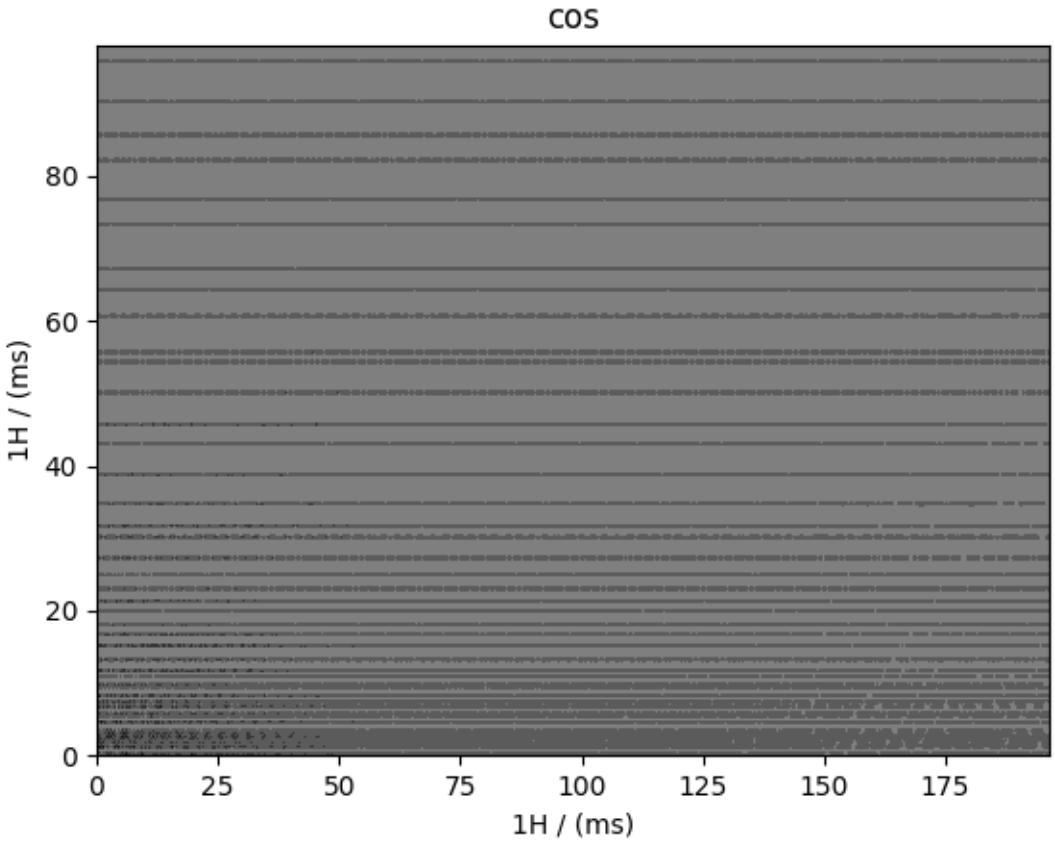
Converting the coordinates to *ms*.

```
x[0].to("ms")
x[1].to("ms")
```

Visualizing the dataset

```
import matplotlib.pyplot as plt

plt.contourf(
    x[0].coordinates.value,
    x[1].coordinates.value,
    y[0].components[0].real,
    cmap="gray_r",
)
plt.xlabel(x[0].axis_label)
plt.ylabel(x[1].axis_label)
plt.title(y[0].name)
plt.show()
```

[1] Balsgart NM, Vosegaard T., Fast Forward Maximum entropy reconstruction of sparsely sampled data., J Magn Reson. 2012, 223, 164-169. doi: 10.1016/j.jmr.2012.07.002



Citation

Total running time of the script: (0 minutes 1.854 seconds)

Sparse along two dimensions, 2D{1,1} dataset

The following is an example[1] of a 2D{1,1} sparse dataset with two-dimensions, $d = 2$, and two, $p = 2$, sparse single-component dependent-variables, where the component is sparsely sampled along two dimensions.

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

```
import csdmpy as cp

filename = "https://osu.box.com/shared/static/kaos28g47brtswi6mgsgaap5qlahp1zo.csdf"
sparse_2d = cp.load(filename)
```

There are two linear dimensions and two single-component sparse dependent variables. The tuple of the dimension and the dependent variable instances are

```
x = sparse_2d.dimensions
y = sparse_2d.dependent_variables
```

[1] Balsgart NM, Vosegaard T., Fast Forward Maximum entropy reconstruction of sparsely sampled data., J Magn Reson. 2012, 223, 164-169. doi: 10.1016/j.jmr.2012.07.002

The coordinates, viewed only for the first ten coordinates, are

```
print(x[0].coordinates[:10])
```

Out:

```
[   0.  192.  384.  576.  768.  960. 1152. 1344. 1536. 1728.] us
```

```
print(x[1].coordinates[:10])
```

Out:

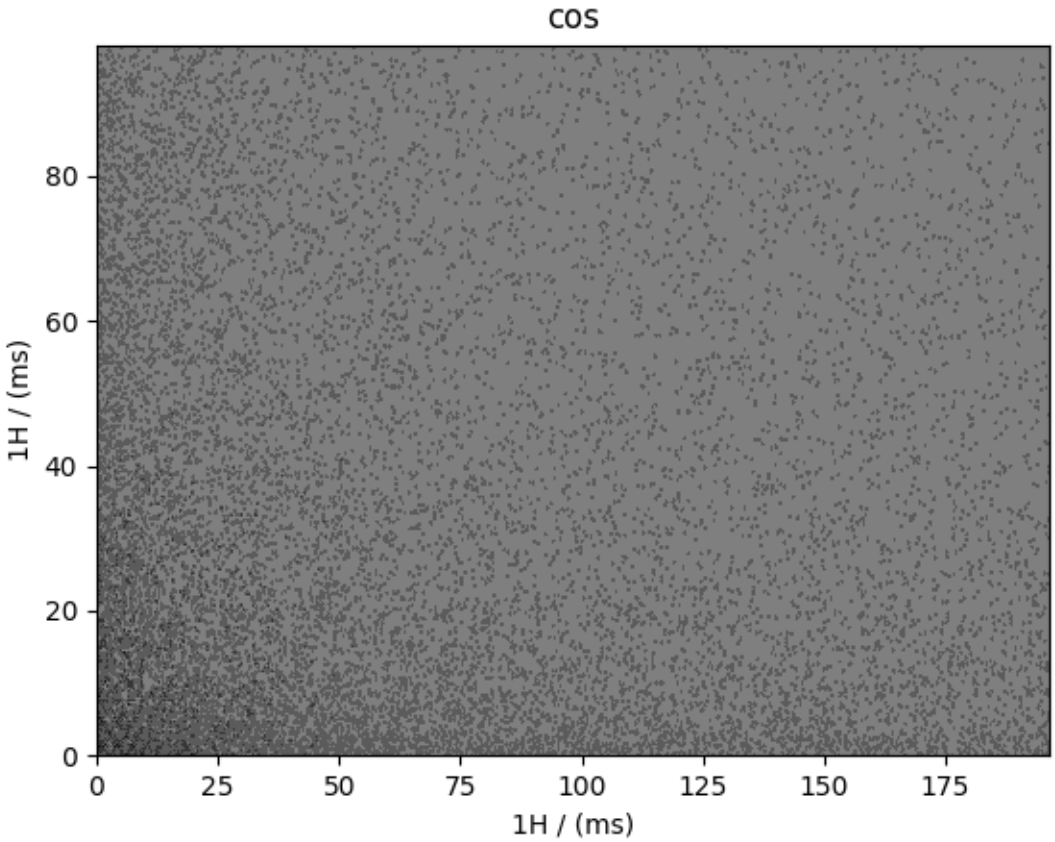
```
[   0.  192.  384.  576.  768.  960. 1152. 1344. 1536. 1728.] us
```

Converting the coordinates to *ms*.

```
x[0].to("ms")
x[1].to("ms")
```

Visualize the dataset

```
import matplotlib.pyplot as plt

plt.contourf(
    x[0].coordinates.value,
    x[1].coordinates.value,
    y[0].components[0].real,
    cmap="gray_r",
)
plt.xlabel(x[0].axis_label)
plt.ylabel(x[1].axis_label)
plt.title(y[0].name)
plt.show()
```



Citation

Total running time of the script: (0 minutes 1.485 seconds)

1.5 Generating csdmpy 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.

1.5.1 Generating Dimension objects

LinearDimension

A LinearDimension is where the coordinates are regularly spaced along the dimension. This type of dimension is frequently encountered in many scientific datasets. There are several ways to generate LinearDimension.

Using the `Dimension` class.

```
>>> import csdmpy as cp
>>> x = cp.Dimension(type='linear', count=10, increment="0.1 s", label="time",␣
↪description="A temporal dimension.")
```

(continues on next page)

(continued from previous page)

```
>>> print(x)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] s)
```

Using the `LinearDimension` class.

```
>>> import csdmpy as cp
>>> x1 = cp.LinearDimension(count=10, increment="0.1 s", label="time",
...                         description="A temporal dimension.")
>>> print(x1)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] s)
```

Using NumPy array

You may also create a LinearDimesion object from a one-dimensional NumPy array using the `as_dimension()` method.

```
>>> import numpy as np
>>> array = np.arange(10) * 0.1
>>> x2 = cp.as_dimension(array)
>>> print(x2)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9])
```

Note, the Dimension object `x2` is dimensionless. You can create a physical dimension by either providing an appropriate unit as the argument to the `as_dimension()` method,

```
>>> x3 = cp.as_dimension(array, unit='s')
>>> print(x3)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] s)
```

or appropriately multiplying the dimension object `x2` with a `ScalarQuantity`.

```
>>> x2 *= cp.ScalarQuantity('s')
>>> print(x2)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] s)
```

The coordinates of the `x2` LinearDimension object, .. doctest:

```
>>> x2.coordinates
<Quantity [0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] s>
```

where `x2.coordinates` is a Quantity array. The value and the unit of the quantity instance are

```
>>> # To access the numpy array
>>> numpy_array = x.coordinates.value
>>> print('numpy array =', numpy_array)
numpy array = [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]

>>> # To access the astropy.unit
>>> unit = x.coordinates.unit
>>> print('unit =', unit)
unit = s
```

respectively.

Note: When generating LinearDimension objects from NumPy array, the NumPy array must be one-dimensional and regularly spaced.

```
>>> cp.as_dimension(np.arange(20).reshape(2, 10))
ValueError: Cannot convert a 2 dimensional array to a Dimension object.
```

MonotonicDimension

A MonotonicDimension is one where the coordinates along the dimension are sampled monotonically, that is, either strictly increasing or decreasing coordinates. Like the LinearDimension, there are several ways to generate a MonotonicDimension.

Using the `Dimension` class.

```
>>> import csdmpy as cp
>>> x = cp.Dimension(type='monotonic',
...                  coordinates=['10ns', '100ns', '1µs', '10µs', '100µs',
...                               '1ms', '10ms', '100ms', '1s', '10s'])
>>> print(x)
MonotonicDimension([1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.
↪e+10] ns)
```

Using the `MonotonicDimension` class.

```
>>> import numpy as np
>>> array = np.asarray([-0.28758166, -0.22712233, -0.19913859, -0.17235106,
...                     -0.1701172, -0.10372635, -0.01817061, 0.05936719,
...                     0.18141424, 0.34758913])
>>> x = cp.MonotonicDimension(coordinates=array)*cp.ScalarQuantity('cm')
>>> print(x)
MonotonicDimension([-0.28758166 -0.22712233 -0.19913859 -0.17235106 -0.1701172  -0.
↪10372635
 -0.01817061  0.05936719  0.18141424  0.34758913] cm)
```

In the above example, we generate a dimensionless MonotonicDimension from the NumPy array and scale its dimensionality by multiplying the object with an appropriate `ScalarQuantity`.

From numpy arrays.

Use the `as_dimension()` method to convert a numpy array as a Dimension object.

```
>>> numpy_array = 10 ** (np.arange(10)/10)
>>> x_dim = cp.as_dimension(numpy_array, unit='A')
>>> print(x_dim)
MonotonicDimension([1.         1.25892541 1.58489319 1.99526231 2.51188643 3.16227766
 3.98107171 5.01187234 6.30957344 7.94328235] A)
```

When generating MonotonicDimension object using the Numpy array, the array must be monotonic, that is, either strictly increasing or decreasing. An exception will be raised otherwise.

```
>>> numpy_array = np.random.rand(10)
>>> x_dim = cp.as_dimension(numpy_array)
Exception: Invalid array for Dimension object.
```

LabeledDimension

A LabeledDimension is one where the coordinates along the dimension are string labels. You can similarly generate a labeled dimension.

Using the `Dimension` class.

```
>>> import csdmpy as cp
>>> x = cp.Dimension(type='labeled',
...                  labels=['The', 'great', 'circle'])
>>> print(x)
LabeledDimension(['The' 'great' 'circle'])
```

Using the `LabeledDimension` class.

```
>>> x = cp.LabeledDimension(labels=['The', 'great', 'circle'])
>>> print(x)
LabeledDimension(['The' 'great' 'circle'])
```

From numpy arrays or python list.

Use the `as_dimension()` method to convert a numpy array as a Dimension object.

```
>>> array = ['The', 'great', 'circle']
>>> x = cp.as_dimension(array)
>>> print(x)
LabeledDimension(['The' 'great' 'circle'])
```

1.5.2 Generating DependentVariable objects

A DependentVariable is where the responses of the multi-dimensional dataset reside. There are two types of DependentVariable objects, *internal* and *external*. In this section, we show how to generate DependentVariable objects of both types.

InternalDependentVariable

Single component dependent variable

Using the `DependentVariable` class.

```
>>> dv1 = cp.DependentVariable(type='internal', quantity_type='scalar',
...                            components=np.arange(10000), unit='J',
...                            description='A sample internal dependent variable.')
>>> print(dv1)
DependentVariable(
[[   0    1    2 ... 9997 9998 9999]] J, quantity_type=scalar, numeric_type=int64)
```

Using NumPy array

Use the `as_dependent_variable()` method to convert a NumPy array into a DependentVariable object. Note, this method returns a view of the NumPy array as the DependentVariable object.

```
>>> dv1 = cp.as_dependent_variable(np.arange(10000).astype(np.complex64), unit='J')
>>> print(dv1)
DependentVariable(
```

(continues on next page)

(continued from previous page)

```
[[0.000e+00+0.j 1.000e+00+0.j 2.000e+00+0.j ... 9.997e+03+0.j
  9.998e+03+0.j 9.999e+03+0.j]] J, quantity_type=scalar, numeric_type=complex64)
```

You may additionally provide the quantity_type for the dependent variable,

```
>>> dv2 = cp.as_dependent_variable(np.arange(10000).astype(np.complex64), quantity_
↪type='pixel_1')
>>> print(dv2)
DependentVariable(
[[0.000e+00+0.j 1.000e+00+0.j 2.000e+00+0.j ... 9.997e+03+0.j
  9.998e+03+0.j 9.999e+03+0.j]], quantity_type=pixel_1, numeric_type=complex64)
```

Multi-component dependent variable

To generate a multi-component DependentVariable object, add an appropriate *quantity_type* value, see *QuantityType* for details.

Using the `DependentVariable` class.

```
>>> dv1 = cp.DependentVariable(type='internal', quantity_type='vector_2',
...                            components=np.arange(10000), unit='J',
...                            description='A sample internal dependent variable.')
>>> print(dv1)
DependentVariable(
[[   0    1    2 ... 4997 4998 4999]
 [5000 5001 5002 ... 9997 9998 9999]] J, quantity_type=vector_2, numeric_type=int64)
```

The above example generates a two-component dependent variable.

Using NumPy array

```
>>> dv1 = cp.as_dependent_variable(np.arange(9000).astype(np.complex64),
...                                unit='m/s', quantity_type='symmetric_matrix_3')
>>> print(dv1)
DependentVariable(
[[0.000e+00+0.j 1.000e+00+0.j 2.000e+00+0.j ... 1.497e+03+0.j
  1.498e+03+0.j 1.499e+03+0.j]
 [1.500e+03+0.j 1.501e+03+0.j 1.502e+03+0.j ... 2.997e+03+0.j
  2.998e+03+0.j 2.999e+03+0.j]
 [3.000e+03+0.j 3.001e+03+0.j 3.002e+03+0.j ... 4.497e+03+0.j
  4.498e+03+0.j 4.499e+03+0.j]
 [4.500e+03+0.j 4.501e+03+0.j 4.502e+03+0.j ... 5.997e+03+0.j
  5.998e+03+0.j 5.999e+03+0.j]
 [6.000e+03+0.j 6.001e+03+0.j 6.002e+03+0.j ... 7.497e+03+0.j
  7.498e+03+0.j 7.499e+03+0.j]
 [7.500e+03+0.j 7.501e+03+0.j 7.502e+03+0.j ... 8.997e+03+0.j
  8.998e+03+0.j 8.999e+03+0.j]] m / s, quantity_type=symmetric_matrix_3, numeric_
↪type=complex64)
```

The above example generates a six-component dependent variable.

Note: For multi-component DependentVariable objects, the size of the NumPy array must be an integer multiple of the total number of components.

```
>>> d1 = cp.as_dependent_variable(np.arange(127), quantity_type='pixel_2')
ValueError: cannot reshape array of size 127 into shape (2,63)
```

Notice in the above examples, we use a one-dimensional NumPy array to generate a DependentVariable object. If a multi-dimensional NumPy array is given as the argument, the array will be raveled (flattened) before returning the DependentVariable object. Note, in the core scientific dataset model, the DependentVariable objects only contain information about the number of components and not the dimensions. For example, consider the following.

```
>>> d2 = cp.as_dependent_variable(np.arange(6000).reshape(10,20,30), quantity_type=
↪'vector_2')
>>> print(d2)
DependentVariable(
[[   0    1    2 ... 2997 2998 2999]
 [3000 3001 3002 ... 5997 5998 5999]], quantity_type=vector_2, numeric_type=int64)
```

Here, a three-dimensional Numpy array is given as the argument with a quantity_type of *vector_2*. The DependentVariable object generated from this array contains two-components by appropriately flattening the input array.

ExternalDependentVariable

The ExternalDependentVariable objects are generated similar to the InternalDependentVariable object. The only difference is that the components of the dependent variable are located at a remote and local address.

Using the `DependentVariable` class.

```
>>> dv = cp.DependentVariable(type='external', quantity_type='scalar', unit='J',
...                           components_url='address to the binary file.',
...                           numeric_type='int64',
...                           description='A sample internal dependent variable.')
```

A DependentVariable of type *external* is useful for data serialization. When using with *csdmpy*, all instances of the *external* dependent variable objects are set as *internal* after downloading the components from the *components_url*.

1.5.3 Generating CSDM objects

An empty csdm object

To create a new empty csdm object, 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": []
```

(continues on next page)

(continued from previous page)

```
  }
}
```

From a NumPy array

Perhaps the easiest way to generate a csdm object is to convert the NumPy array holding the dataset as a csdm object using the `as_csdm()` method, which returns a view of the array as a CSDM object. Here, the NumPy array becomes the dependent variable of the CSDM object of the given *quantity_type*. Unlike the `as_dependent_variable()` method, however, the `as_csdm()` method retains the shape of the Numpy array and uses this information to generate the dimensions of the CSDM object. By default, the dimensions are of a *linear* subtype with unit increment. Consider the following example.

```
>>> array = np.arange(30).reshape(3, 10)
>>> csdm_obj = cp.as_csdm(array)
>>> print(csdm_obj)
CSDM(
DependentVariable(
[[[ 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]]], quantity_type=scalar, numeric_type=int64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]),
LinearDimension([0. 1. 2.])
)
```

Here, a two-dimensional NumPy array of shape (3, 10) is given as the argument of the `as_csdm()` method. The resulting CSDM object, `csdm_obj`, contains a 2D{1} datasets, with two linear dimensions of unit increment and 10 and 3 points, respectively, and a single one-component dependent variable of quantity_type *scalar*.

Note: The order of the dimensions in the CSDM object is the reverse of the order of axes from the corresponding Numpy array. Thus, the dimension at index 0 of the CSDM object is the last axis of the Numpy array.

You may additionally provide a quantity type as the argument of the `as_csdm()` method. When the quantity type requires more than one component, see *QuantityType*, the first axis of the NumPy array must be the number of components. For example,

```
>>> csdm_obj1 = cp.as_csdm(array, quantity_type='pixel_3')
>>> print(csdm_obj1)
CSDM(
DependentVariable(
[[ 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]], quantity_type=pixel_3, numeric_type=int64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.])
)
```

Here, the `csdm_obj1` object is a 1D{3} datasets, with a single three-component dependent variable. In this case, the length of the NumPy array along axis 0, i.e., 3, is consistent with the number of components required by the quantity type *pixel_3*. The remaining axes of the NumPy array are used in generating the dimensions of the csdm object. In this example, this corresponds to a single dimension of *linear* type with 10 points.

The following example generates a 3D{2} vector dataset. Here, the first axis of the four-dimensional Numpy array is the components of the vector dataset, and the remaining three axes become the respective dimensions.

```
>>> array2 = np.arange(12000).reshape(2,30,20,10)
>>> csdm_obj2 = cp.as_csdm(array2, quantity_type='vector_2')
>>> print(len(csdm_obj2.dimensions), len(csdm_obj2.dependent_variables[0].components))
3 2
```

An exception will be raised if the *quantity_type* and the number of points along the first axis of the NumPy array are inconsistent, for example,

```
>>> csdm_obj_err = cp.as_csdm(array, quantity_type='vector_2')
ValueError: Expecting exactly 2 components for quantity type, `vector_2`, found 3.
Make sure `array.shape[0]` is equal to the number of components supported by vector_2.
```

Note: Only a csdm object with a single dependent variable may be created from a NumPy array.

1.5.4 Adding instances of Dimension class to CSDM object

Create a new empty CSDM object following,

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

Add an instance of the Dimension class using the `add_dimension()` method of the *CSDM* instance. There are three subtypes of Dimension objects,

- LinearDimension
- MonotonicDimension
- LabeledDimension

Using an instance of the Dimension class

Please read the topic *Generating Dimension objects* for details on how to generate an instance of the Dimension class. Once created, use the `add_dimension()` method of the CSDM object to add the dimension, for example,

```
>>> linear_dim = cp.LinearDimension(count=10, increment='0.1 C/V')
>>> new_data.add_dimension(linear_dim)
>>> print(new_data)
CSDM(
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] C / V)
)
```

Using Python's dictionary objects

When using python dictionaries, the key-value pairs of the dictionary must be a valid collection for the given Dimension subtype. For example,

```
>>> # dictionary representation of a linear dimension.
>>> d0 = {
...     'type': 'linear',
...     'description': 'This is a linear dimension',
...     'count': 5,
...     'increment': '0.1 rad'
... }
>>> # dictionary representation of a monotonic dimension.
```

(continues on next page)

(continued from previous page)

```
>>> d1 = {
...     'type': 'monotonic',
...     'description': 'This is a monotonic dimension',
...     'coordinates': ['1 m/s', '2 cm/s', '4 mm/s'],
... }
>>> # dictionary representation of a labeled dimension.
>>> d2 = {
...     'type': 'labeled',
...     'description': 'This is a labeled dimension',
...     'labels': ['Cu', 'Ag', 'Au'],
... }
>>> # add the dictionaries to the CSDM object.
>>> new_data.add_dimension(d0)
>>> new_data.add_dimension(d1)
>>> new_data.add_dimension(d2)
>>> print(new_data)
CSDM(
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] C / V),
LinearDimension([0.  0.1 0.2 0.3 0.4] rad),
MonotonicDimension([1.    0.02  0.004] m / s),
LabeledDimension(['Cu' 'Ag' 'Au'])
)
```

Attention: When using the *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 `linear_dim`, in the above example, will affect the corresponding dimension instance in the `new_data` instance. To avoid this, you may pass a copy of the instance, `linear_dim.copy()`, as the argument to the `add_dimension()` method.

1.5.5 Adding instances of DependentVariable class to CSDM object

Create a new empty CSDM object following,

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

Add an instance of the DependentVariable class 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.

Using an instance of the DependentVariable class

Please read the topic *Generating DependentVariable objects* for details on how to generate an instance of the DependentVariable class. Once created, use the `add_dependent_variable()` method of the CSDM object to add the dependent variable, for example,

```
>>> dv = cp.as_dependent_variable(np.arange(10))
>>> new_data.add_dependent_variable(dv)
```

(continues on next page)

(continued from previous page)

```
>>> print(new_data)
CSDM(
DependentVariable(
[[0 1 2 3 4 5 6 7 8 9]], quantity_type=scalar, numeric_type=int64)
)
```

Using Python's dictionary objects

When using python dictionaries, the key-value pairs of the dictionary must be a valid collection for the given DependentVariable subtype. For example,

```
>>> d0 = {
...     'type': 'internal',
...     'quantity_type': 'scalar',
...     'description': 'This is an internal scalar dependent variable',
...     'unit': 'cm',
...     'components': np.arange(100)
... }
>>> new_data.add_dependent_variable(d0)
>>> print(new_data)
CSDM(
DependentVariable(
[[0 1 2 3 4 5 6 7 8 9]], quantity_type=scalar, numeric_type=int64),
DependentVariable(
[[ 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 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 61 62 63 64 65 66 67 68 69 70 71
  72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
  96 97 98 99]] cm, quantity_type=scalar, numeric_type=int64)
)
```

1.5.6 Tutorial examples on generating csdm datasets

2D{1} Datasets

Linear and Monotonic dimensions

In the following example, we illustrate how one can covert a Numpy array into a CSDM object. Start by importing the Numpy and csdmpy libraries.

```
import numpy as np

import csdmpy as cp
```

Let's generate a 2D NumPy array of random numbers as our dataset.

```
data = np.random.rand(8192).reshape(32, 256)
```

To convert this array into a csdm object, use the *as_csdm()* method,

```
data_csdm = cp.as_csdm(data)
print(data_csdm.dimensions)
```

Out:

```
[LinearDimension(count=256, increment=1.0),
LinearDimension(count=32, increment=1.0)]
```

This generates a 2D{1} dataset, that is, a two-dimensional dataset with a single one-component dependent variable. The two dimensions are, by default, set as the LinearDimensions of the unit interval.

You may set the proper dimensions by generating the appropriate Dimension objects and replacing the default dimensions in the `data_csdm` object.

```
d0 = cp.LinearDimension(
    count=256, increment="15.23 µs", coordinates_offset="-1.95 ms", label="t1"
)
```

Here, `d0` is a LinearDimension with 256 points and 15.23 µs increment. You may similarly set the second dimension as a LinearDimension, however, in this example, let's set it as a MonotonicDimension.

```
array = 10 ** (np.arange(32) / 8)
d1 = cp.as_dimension(array, unit="µs", label="t2")
```

The variable `array` is a NumPy array that is uniformly sampled on a log scale. To convert this array into a Dimension object, we use the *`as_dimension()`* method.

Now, replace the dimension objects in `data_csdm` with the new ones.

```
data_csdm.dimensions[0] = d0
data_csdm.dimensions[1] = d1
```

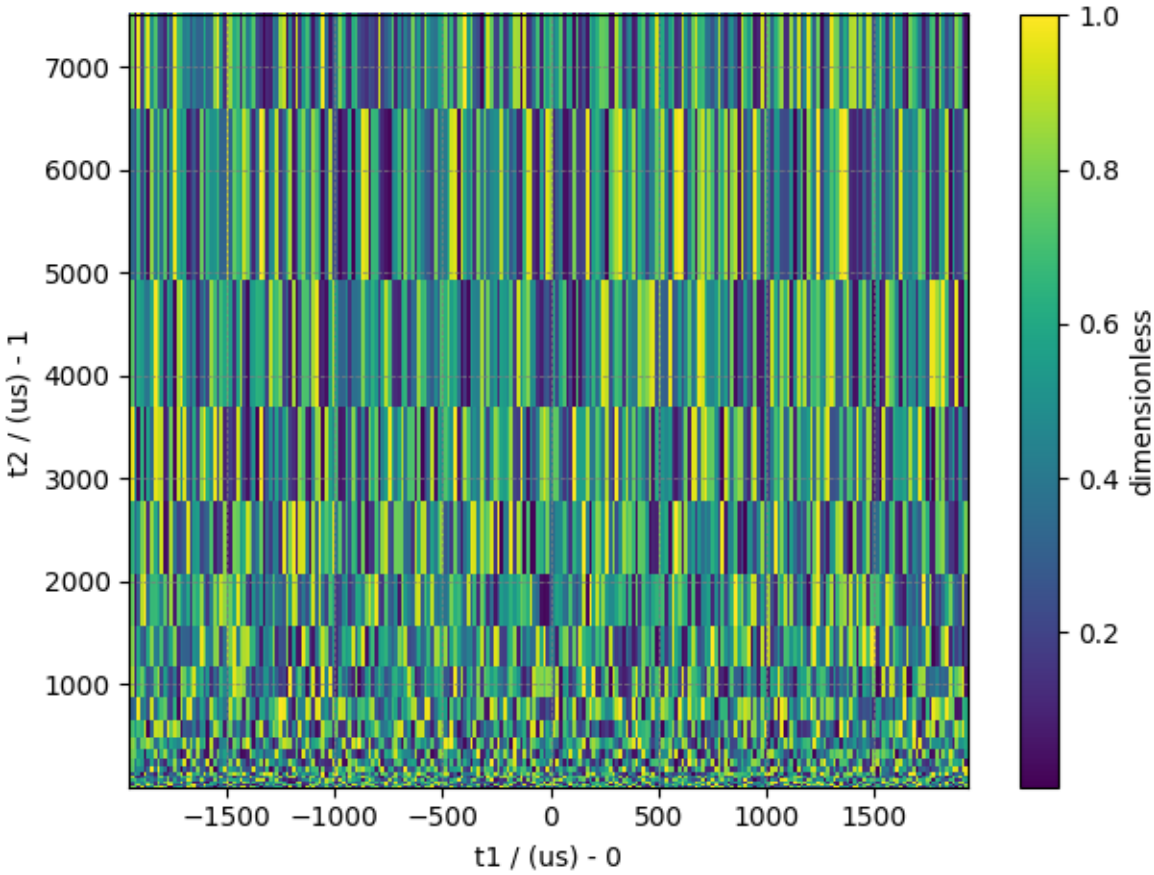
```
print(data_csdm.dimensions)
```

Out:

```
[LinearDimension(count=256, increment=15.23 µs, coordinates_offset=-1.95 ms, quantity_
↪name=time, label=t1, reciprocal={'quantity_name': 'frequency'}),
MonotonicDimension(coordinates=[1.00000000e+00 1.33352143e+00 1.77827941e+00 2.
↪37137371e+00
 3.16227766e+00 4.21696503e+00 5.62341325e+00 7.49894209e+00
 1.00000000e+01 1.33352143e+01 1.77827941e+01 2.37137371e+01
 3.16227766e+01 4.21696503e+01 5.62341325e+01 7.49894209e+01
 1.00000000e+02 1.33352143e+02 1.77827941e+02 2.37137371e+02
 3.16227766e+02 4.21696503e+02 5.62341325e+02 7.49894209e+02
 1.00000000e+03 1.33352143e+03 1.77827941e+03 2.37137371e+03
 3.16227766e+03 4.21696503e+03 5.62341325e+03 7.49894209e+03] us, quantity_name=time,␣
↪label=t2, reciprocal={'quantity_name': 'frequency'})]
```

Plot of the dataset.

```
cp.plot(data_csdm)
```



To serialize the file, use the save method.

```
data_csdm.save("filename.csdf")
```

Total running time of the script: (0 minutes 0.235 seconds)

1.6 Interacting with csdmpy objects

1.6.1 Interacting with Dimension objects

LinearDimension

There are several attributes and methods associated with the LinearDimension, each controlling the coordinates along the dimension. The following section demonstrates the effect of these attributes and methods on the coordinates of the LinearDimension.

```
>>> import csdmpy as cp
>>> x = cp.LinearDimension(count=10, increment="0.1 s", label="time",
...                        description="A temporal dimension.")
>>> print(x)
LinearDimension([0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9] s)
```

Attributes

type This attribute returns the type of the instance.

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

The attributes that modify the coordinates

count The number of points along the dimension

```
>>> print('number of points =', x.count)
number of points = 10
```

To update the number of points, update the value of this attribute,

```
>>> x.count = 12
>>> print('new number of points =', x.count)
new number of points = 12

>>> print('new coordinates =', x.coordinates)
new coordinates = [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1] s
```

increment

```
>>> print('old increment =', x.increment)
old increment = 0.1 s

>>> x.increment = "10 s"
>>> print('new increment =', x.increment)
new increment = 10.0 s

>>> print('new coordinates =', x.coordinates)
new coordinates = [  0.  10.  20.  30.  40.  50.  60.  70.  80.  90. 100. 110.] s
```

coordinates_offset

```
>>> print('old reference offset =', x.coordinates_offset)
old reference offset = 0.0 s

>>> x.coordinates_offset = "1 s"
>>> print('new reference offset =', x.coordinates_offset)
new reference offset = 1.0 s

>>> print('new coordinates =', x.coordinates)
new coordinates = [  1.  11.  21.  31.  41.  51.  61.  71.  81.  91. 101. 111.] s
```

origin_offset

```
>>> print('old origin offset =', x.origin_offset)
old origin offset = 0.0 s

>>> x.origin_offset = "1 day"
>>> print ('new origin offset =', x.origin_offset)
new origin offset = 1.0 d

>>> print('new coordinates =', x.coordinates)
new coordinates = [  1.  11.  21.  31.  41.  51.  61.  71.  81.  91. 101. 111.] s
```

The last operation updates the value of the origin offset, however, the coordinates remain unaffected. This is because the *coordinates* attribute refers to the reference coordinates. You may access the absolute coordinates through the *absolute_coordinates* attribute.

```
>>> print('absolute coordinates =', x.absolute_coordinates)
absolute coordinates = [86401. 86411. 86421. 86431. 86441. 86451. 86461. 86471.␣
↪86481. 86491.
 86501. 86511.] s
```

The attributes that modify the order of coordinates

complex_fft If true, orders the coordinates along the dimension according to the output of a complex Fast Fourier Transform (FFT) routine.

```
>>> print('old coordinates =', x.coordinates)
old coordinates = [  1.  11.  21.  31.  41.  51.  61.  71.  81.  91. 101. 111.] s

>>> x.complex_fft = True
>>> print('new coordinates =', x.coordinates)
new coordinates = [-59. -49. -39. -29. -19.  -9.   1.  11.  21.  31.  41.  51.] s
```

Other attributes

period The period of the dimension.

```
>>> print('old period =', x.period)
old period = inf s

>>> x.period = '10 s'
>>> print('new period =', x.period)
new period = 10.0 s
```

quantity_name Returns the quantity name.

```
>>> print('quantity name is', x.quantity_name)
quantity name is time
```

label

```
>>> x.label
'time'

>>> x.label = 't1'
>>> x.label
't1'
```

axis_label Returns a formatted string for labeling axis.

```
>>> x.label
't1'
>>> x.axis_label
't1 / (s)'
```

Methods

`to()`: This method is used for unit conversions.

```
>>> print('old unit =', x.coordinates.unit)
old unit = s

>>> print('old coordinates =', x.coordinates)
old coordinates = [-59. -49. -39. -29. -19.  -9.   1.  11.  21.  31.  41.  51.] s

>>> ## unit conversion
>>> x.to('min')

>>> print ('new coordinates =', x.coordinates)
new coordinates = [-0.98333333 -0.81666667 -0.65       -0.48333333 -0.31666667 -0.15
  0.01666667  0.18333333  0.35        0.51666667  0.68333333  0.85      ] min
```

Note: In the above examples, the coordinates are ordered according to the FFT output order, based on the previous set of operations.

The argument of this method is a string containing the unit, in this case, *min*, whose dimensionality is be consistent with the dimensionality of the coordinates. An exception will be raised otherwise.

```
>>> x.to('km/s')
Exception: The unit 'km / s' (speed) is inconsistent with the unit 'min' (time).
```

Changing the dimensionality

You may scale the dimension object by multiplying the object with the appropriate ScalarQuantity, as follows,

```
>>> print(x)
LinearDimension([-0.98333333 -0.81666667 -0.65       -0.48333333 -0.31666667 -0.15
  0.01666667  0.18333333  0.35        0.51666667  0.68333333  0.85      ] min)
>>> x *= cp.ScalarQuantity('m/s')
>>> print(x)
LinearDimension([-59. -49. -39. -29. -19.  -9.   1.  11.  21.  31.  41.  51.] m)
```

MonotonicDimension

There are several attributes and methods associated with a MonotonicDimension, controlling the coordinates along the dimension. The following section demonstrates the effect of these attributes and methods on the coordinates.

```
>>> import numpy as np
>>> array = np.asarray([-0.28758166, -0.22712233, -0.19913859, -0.17235106,
...                     -0.1701172, -0.10372635, -0.01817061, 0.05936719,
...                     0.18141424, 0.34758913])
>>> x = cp.MonotonicDimension(coordinates=array)*cp.ScalarQuantity('cm')
```

Attributes

The following are the attributes of the `MonotonicDimension` instance.

type This attribute returns the type of the instance.

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

The attributes that modify the coordinates

count The number of points along the dimension

```
>>> print ('number of points =', x.count)
number of points = 10
```

You may update the number of points with this attribute, however, you can only lower the number of points.

```
>>> x.count = 6
>>> print('new number of points =', x.count)
new number of points = 6
>>> print(x.coordinates)
[-0.28758166 -0.22712233 -0.19913859 -0.17235106 -0.1701172  -0.10372635] cm
```

origin_offset

```
>>> print('old origin offset =', x.origin_offset)
old origin offset = 0.0 cm

>>> x.origin_offset = "1 km"
>>> print('new origin offset =', x.origin_offset)
new origin offset = 1.0 km

>>> print(x.coordinates)
[-0.28758166 -0.22712233 -0.19913859 -0.17235106 -0.1701172  -0.10372635] cm
```

The last operation updates the value of the origin offset, however, the value of the `coordinates` attribute remains unchanged. This is because the `coordinates` refer to the reference coordinates. The absolute coordinates are accessed through the `absolute_coordinates` attribute.

```
>>> print('absolute coordinates =', x.absolute_coordinates)
absolute coordinates = [99999.71241834 99999.77287767 99999.80086141 99999.
↪82764894
 99999.8298828  99999.89627365] cm
```

Other attributes

label

```
>>> x.label = 't1'
>>> print('new label =', x.label)
new label = t1
```

period

```
>>> print('old period =', x.period)
old period = inf cm
```

(continues on next page)

(continued from previous page)

```
>>> x.period = '10 m'
>>> print('new period =', x.period)
new period = 10.0 m
```

quantity_name Returns the quantity name.

```
>>> print ('quantity is', x.quantity_name)
quantity is length
```

Methods

to()

The method is used for unit conversions. It follows,

```
>>> print('old unit =', x.coordinates.unit)
old unit = cm
>>> print('old coordinates =', x.coordinates)
old coordinates = [-0.28758166 -0.22712233 -0.19913859 -0.17235106 -0.1701172  -0.
↪10372635] cm

>>> ## unit conversion
>>> x.to('mm')

>>> print('new coordinates =', x.coordinates)
new coordinates = [-2.8758166 -2.2712233 -1.9913859 -1.7235106 -1.701172  -1.0372635]␣
↪mm
```

The argument of this method is a unit, in this case, 'mm', whose dimensionality must be consistent with the dimensionality of the coordinates. An exception will be raised otherwise,

```
>>> x.to('km/s')
Exception("Validation Failed: The unit 'km / s' (speed) is inconsistent with the unit
↪'mm' (length).")
```

Changing the dimensionality

You may scale the dimension object by multiplying the object with the appropriate ScalarQuantity, as follows,

```
>>> print(x)
MonotonicDimension([-2.8758166 -2.2712233 -1.9913859 -1.7235106 -1.701172  -1.
↪0372635] mm)
>>> x *= cp.ScalarQuantity('2 s/mm')
>>> print(x)
MonotonicDimension([-0.57516332 -0.45424466 -0.39827718 -0.34470212 -0.3402344  -0.
↪2074527 ] cm s / mm)
```

1.6.2 Interacting with CSDM objects

Basic math operations

The csdm object supports basic mathematical operations such as additive and multiplicative operations.

Note: All operations applied to or involving the csdm objects apply only to the components of the dependent variables within the csdm object. These operations do not apply to the dimensions within the csdm object.

Consider the following csdm data object.

```
>>> arr1 = np.arange(6, dtype=np.float32).reshape(2, 3)
>>> csdm_obj1 = cp.as_csdm(arr1)
>>> # converting the dimension to proper physical dimensions.
>>> csdm_obj1.dimensions[0]*=cp.ScalarQuantity('2.64 m')
>>> csdm_obj1.dimensions[0].coordinates_offset = '1 km'
>>> # converting the dimension to proper physical dimensions.
>>> csdm_obj1.dimensions[1]*=cp.ScalarQuantity('10 µs')
>>> csdm_obj1.dimensions[1].coordinates_offset = '-0.5 ms'
>>> print(csdm_obj1)
CSDM(
DependentVariable(
[[[0. 1. 2.]
  [3. 4. 5.]]], quantity_type=scalar, numeric_type=float32),
LinearDimension([1000.   1002.64 1005.28] m),
LinearDimension([-500. -490.] us)
)
```

Additive operations involving a scalar

Example 1

```
>>> csdm_obj1 += np.pi
>>> print(csdm_obj1)
CSDM(
DependentVariable(
[[[3.1415927 4.141593  5.141593 ]
  [6.141593  7.141593  8.141593 ]]], quantity_type=scalar, numeric_type=float32),
LinearDimension([1000.   1002.64 1005.28] m),
LinearDimension([-500. -490.] us)
)
```

Example 2

```
>>> csdm_obj2 = csdm_obj1 + (2 - 4j)
>>> print(csdm_obj2)
CSDM(
DependentVariable(
[[[ 5.141593-4.j  6.141593-4.j  7.141593-4.j]
  [ 8.141593-4.j  9.141593-4.j 10.141593-4.j]]], quantity_type=scalar, numeric_
↪type=complex64),
LinearDimension([1000.   1002.64 1005.28] m),
LinearDimension([-500. -490.] us)
)
```

Multiplicative operations involving scalar / ScalarQuantity

Example 3

```
>>> csdm_obj1 = cp.as_csdm(np.ones(6).reshape(2, 3))
>>> csdm_obj2 = csdm_obj1 * 4.693
>>> print(csdm_obj2)
CSDM(
DependentVariable(
[[[4.693 4.693 4.693]
  [4.693 4.693 4.693]]], quantity_type=scalar, numeric_type=float64),
LinearDimension([0. 1. 2.]),
LinearDimension([0. 1.])
)
```

Example 4

```
>>> csdm_obj2 = csdm_obj1 * 3j/2.4
>>> print(csdm_obj2)
CSDM(
DependentVariable(
[[[0.+1.25j 0.+1.25j 0.+1.25j]
  [0.+1.25j 0.+1.25j 0.+1.25j]]], quantity_type=scalar, numeric_type=complex128),
LinearDimension([0. 1. 2.]),
LinearDimension([0. 1.])
)
```

You may change the dimensionality of the dependent variables by multiplying the csdm object with the appropriate scalar quantity, for example,

Example 5

```
>>> csdm_obj1 *= cp.ScalarQuantity('3.23 m')
>>> print(csdm_obj1)
CSDM(
DependentVariable(
[[[3.23 3.23 3.23]
  [3.23 3.23 3.23]]] m, quantity_type=scalar, numeric_type=float64),
LinearDimension([0. 1. 2.]),
LinearDimension([0. 1.])
)
```

Example 6

```
>>> csdm_obj1 /= cp.ScalarQuantity('3.23 m')
>>> print(csdm_obj1)
CSDM(
DependentVariable(
[[[1. 1. 1.]
  [1. 1. 1.]]], quantity_type=scalar, numeric_type=float64),
LinearDimension([0. 1. 2.]),
LinearDimension([0. 1.])
)
```

Additive operations involving two csdm objects

The additive operations are supported between two csdm objects only when the two objects have identical sets of Dimension objects and DependentVariable objects with the same dimensionality. For examples,

Example 7

```
>>> csdm1 = cp.as_csdm(np.ones((2,3)), unit='m/s')
>>> csdm2 = cp.as_csdm(np.ones((2,3)), unit='cm/s')
>>> csdm_obj = csdm1 + csdm2
>>> print(csdm_obj)
CSDM(
DependentVariable(
[[[1.01 1.01 1.01]
  [1.01 1.01 1.01]]] m / s, quantity_type=scalar, numeric_type=float64),
LinearDimension([0. 1. 2.]),
LinearDimension([0. 1.])
)
```

An exception will be raised if the DependentVariable objects of the two csdm objects have different dimensionality.

Example 8

```
>>> csdm1 = cp.as_csdm(np.ones((2,3)), unit='m/s')
>>> csdm2 = cp.as_csdm(np.ones((2,3)))
>>> csdm_obj = csdm1 + csdm2
Exception: Cannot operate on dependent variables with physical types: speed and␣
↪dimensionless.
```

Similarly, an exception will be raised if the dimension objects of the two csdm objects are different.

Example 9

```
>>> csdm1 = cp.as_csdm(np.ones((2,3)), unit='m/s')
>>> csdm1.dimensions[1] = cp.MonotonicDimension(coordinates=['1 ms', '1 s'])
>>> csdm2 = cp.as_csdm(np.ones((2,3)), unit='cm/s')
>>> csdm_obj = csdm1 + csdm2
Exception: Cannot operate on CSDM objects with different dimensions.
```

Basic Slicing and Indexing

The CSDM objects support NumPy basic slicing and indexing and follow the same rules as the NumPy array. Consider the following 3D{1} csdm object.

```
>>> csdm1 = cp.as_csdm(np.zeros((5, 10, 20)), unit='s')
>>> csdm1.dimensions[0] = cp.as_dimension(np.arange(20)*0.5+4.3, unit='kg')
>>> csdm1.dimensions[1] = cp.as_dimension([1, 2, 3, 5, 7, 11, 13, 17, 19, 23], unit=
↪'mm')
>>> csdm1.dimensions[2] = cp.LabeledDimension(labels=list('abcde'))
>>> print(csdm1.shape)
(20, 10, 5)
>>> print(csdm1.dimensions)
[LinearDimension(count=20, increment=0.5 kg, coordinates_offset=4.3 kg, quantity_
↪name=mass),
MonotonicDimension(coordinates=[ 1.  2.  3.  5.  7. 11. 13. 17. 19. 23.] mm, quantity_
↪name=length, reciprocal={'quantity_name': 'wavenumber'}),
LabeledDimension(labels=['a', 'b', 'c', 'd', 'e'])]
```

The above object `csdm1` has three dimensions, each with different dimensionality and dimension type. To retrieve a sub-grid of this 3D{1} dataset, use the NumPy indexing scheme.

Example 10

```
>>> sub_csdm = csdm1[0]
>>> print(sub_csdm.shape)
(10, 5)
>>> print(sub_csdm.dimensions)
[MonotonicDimension(coordinates=[ 1.  2.  3.  5.  7. 11. 13. 17. 19. 23.] mm,␣
↪quantity_name=length, reciprocal={'quantity_name': 'wavenumber'}),
LabeledDimension(labels=['a', 'b', 'c', 'd', 'e'])]
```

The above example returns a 2D{1} cross-section of the 3D{1} datasets corresponding to the index 0 along the first dimension of the `csdm1` object as a `sub_csdm` csdm object. The two dimensions in `sub_csdm` are the MonotonicDimension and LabeledDimension.

Example 11

```
>>> sub_csdm = csdm1[::5, 2::2, :]
>>> print(sub_csdm.shape)
(4, 4, 5)
>>> print(sub_csdm.dimensions)
[LinearDimension(count=4, increment=2.5 kg, coordinates_offset=4.3 kg, quantity_
↪name=mass),
MonotonicDimension(coordinates=[ 3.  7. 13. 19.] mm, quantity_name=length, reciprocal=
↪{'quantity_name': 'wavenumber'}),
LabeledDimension(labels=['a', 'b', 'c', 'd', 'e'])]
```

The above example returns a 3D{1} dataset, `sub_csdm`, which contains a sub-grid of the 3D{1} datasets in `csdm1`. In `sub_csdm`, the first dimension is a sub-grid of the first dimension from the `csdm1` object, where only every fifth grid point is selected. Similarly, the second dimension of the `sub_csdm` object is sampled from the second dimension of the `csdm1` object, where every second grid point is selected, starting with the entry at the grid index two. The third dimension of the `sub_csdm` object is the same as the third object of the `csdm1` object. The values of the corresponding linear, monotonic, and labeled dimensions are accordingly adjusted, for example, notice the value of the *count* and *increment* attribute of the linear dimension in `sub_csdm` object.

Example 12

```
>>> sub_csdm = csdm1[::5, 2::2, -3::-1]
>>> print(sub_csdm.shape)
(4, 4, 3)
>>> print(sub_csdm.dimensions)
[LinearDimension(count=4, increment=2.5 kg, coordinates_offset=4.3 kg, quantity_
↪name=mass),
MonotonicDimension(coordinates=[ 3.  7. 13. 19.] mm, quantity_name=length, reciprocal=
↪{'quantity_name': 'wavenumber'}),
LabeledDimension(labels=['c', 'b', 'a'])]
```

The above example is similar to the previous examples, except the third dimension indexed in reversed starting at the third index from the end.

See also:

Basic Slicing and Indexing

Support for Numpy methods

In most cases, the csdm object may be used as if it were a NumPy array. See the list of all supported *Supported NumPy functions*.

Method that only operate on dimensionless dependent variables

Example 13

```
>>> csdm_obj1 = cp.as_csdm(10**(np.arange(10)/10))
>>> new_csdm1 = np.log10(csdm_obj1)
>>> print(new_csdm1)
CSDM(
DependentVariable(
[[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]], quantity_type=scalar, numeric_
↪type=float64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.])
)
```

Example 14

```
>>> new_csdm2 = np.cos(2*np.pi*new_csdm1)
>>> print(new_csdm2)
CSDM(
DependentVariable(
[[ 1.          0.80901699  0.30901699 -0.30901699 -0.80901699 -1.
  -0.80901699 -0.30901699  0.30901699  0.80901699]], quantity_type=scalar, numeric_
↪type=float64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.])
)
```

Example 15

```
>>> new_csdm2 = np.exp(new_csdm1 * cp.ScalarQuantity('K'))
ValueError: Cannot apply `exp` to quantity with physical type `temperature`.
```

An exception is raised for csdm object with non-dimensionless dependent variables.

Method that are independent of the dependent variable dimensionality

Example 16

```
>>> new_csdm2 = np.square(new_csdm1 * cp.ScalarQuantity('K'))
>>> print(new_csdm2)
CSDM(
DependentVariable(
[[0.   0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81]] K2, quantity_type=scalar,␣
↪numeric_type=float64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.])
)
```

Example 17

```
>>> new_csdm1 = np.sqrt(new_csdm2)
>>> print(new_csdm1)
CSDM(
DependentVariable(
[[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]] K, quantity_type=scalar, numeric_
↪type=float64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.])
)
```

Dimension reduction methods

Example 18

```
>>> csdm1 = cp.as_csdm(np.ones((10,20,30)), unit='µG')
>>> csdm1.shape
(30, 20, 10)
>>> new = np.sum(csdm1, axis=1)
>>> new.shape
(30, 10)
>>> print(new.dimensions)
[LinearDimension(count=30, increment=1.0),
LinearDimension(count=10, increment=1.0)]
```

Example 19

```
>>> csdm1 = cp.as_csdm(np.ones((10,20,30)), unit='µG')
>>> csdm1.shape
(30, 20, 10)
>>> new = np.sum(csdm1, axis=(1, 2))
>>> new.shape
(30,)
>>> print(new.dimensions)
[LinearDimension(count=30, increment=1.0)]
```

Example 20

```
>>> minimum = np.min(new_csdm1)
>>> print(minimum)
0.0 K
>>> np.min(new_csdm1) == new_csdm1.min()
True
```

Note: See the list of all supported *Supported NumPy functions*.

1.7 Serializing CSDM object to files

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.

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.

1.8 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": [
          "⍰",
          "⍰",
          "⍰",
```

(continues on next page)

(continued from previous page)

```
          "⍰",
          "⍰",
          "⍰"
        ]
      }
    ],
    "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')
```

1.9 API-Reference

1.9.1 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.

Methods

Methods Summary

`parse_dict`	Parse a CSDM compliant python dictionary and return a CSDM object.
`load`	Loads a .csdf/.csdfe file and returns an instance of the *CSDM* class.
`loads`	Loads a JSON serialized string as a CSDM object.
`new`	Creates a new instance of the *CSDM* class containing a 0D{0} dataset.
`as_dimension`	Generate and return a Dimension object from a 1D numpy array.
`as_dependent_variable`	Generate and return a DependentVariable object from a 1D or 2D numpy array.
`as_csdm`	Generate and return a view of the nD numpy array as a csdm object.
`plot`	A supplementary function for plotting basic 1D and 2D datasets only.

Method Documentation

`csdmpy.parse_dict`(*dictionary*)

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

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

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

Loads a .csdf/.csdfe file and returns an instance of the *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 a 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.
- **verbose** (*bool*) – If the filename is a URL, this option will show the progress bar for the file download status, when True.

Returns A CSDM instance.

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

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

Parameters **description** (*str*) – A string describing 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.**as_csdm**(*array*, *unit=''*, *quantity_type='scalar'*)

Generate and return a view of the nD numpy array as a csdm object. The nD array is the dependent variable of the csdm object of the given quantity type. The shape of the nD array is used to generate Dimension object of *linear* subtype.

Parameters

- **array** – The nD numpy array.
- **unit** – The unit for the dependent variable. The default is empty string.
- **quantity_type** – The quantity type of the dependent variable.

Example

```
>>> array = np.arange(30).reshape(3, 10)
>>> csdm_obj = cp.as_csdm(array)
>>> print(csdm_obj)
CSDM(
DependentVariable(
[[[ 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]]], quantity_type=scalar, numeric_type=int64),
LinearDimension([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]),
LinearDimension([0. 1. 2.])
)
```

csdmpy.**as_dimension**(*array*, *unit=''*, *type=None*, *label=''*, *description=''*, *application={}*)

Generate and return a Dimension object from a 1D numpy array.

Parameters

- **array** – A 1D numpy array.
- **unit** – The unit of the coordinates along the dimension.
- **type** – The dimension type. Valid values are linear, monotonic, labeled, or None. If the value is None, let us decide. The default value is None.
- **label** – The label along the dimension. The default value is an empty string.
- **description** – A description of the dimension. The default value is an empty string.

- **application** – An application dictionary. The default is an empty dictionary.

Example

```
>>> array = np.arange(15)*0.5
>>> dim_object = cp.as_dimension(array)
>>> print(dim_object)
LinearDimension([0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7. ])
```

```
>>> array = ['The', 'great', 'circle']
>>> dim_object = cp.as_dimension(array, label='in the sky')
>>> print(dim_object)
LabeledDimension(['The' 'great' 'circle'])
```

csdmpy.**as_dependent_variable**(*array*, *quantity_type='scalar'*, *unit=''*, *description=''*, *application={}*)

Generate and return a DependentVariable object from a 1D or 2D numpy array.

Parameters

- **array** – A 1D or 2D numpy array.
- **quantity_type** – The quantity type of the dependent variable. See *QuantityType* for valid quantity types.
- **unit** – The unit of the dependent variable components.
- **label** – The label along the dimension. The default value is an empty string.
- **description** – A description of the dimension. The default value is an empty string.
- **application** – An application dictionary. The default is an empty dictionary.

Example

```
>>> array = np.arange(1e4).astype(np.complex128)
>>> dim_object = cp.as_dependent_variable(array, )
>>> print(dim_object)
DependentVariable(
[[0.000e+00+0.j 1.000e+00+0.j 2.000e+00+0.j ... 9.997e+03+0.j
  9.998e+03+0.j 9.999e+03+0.j]], quantity_type=scalar, numeric_type=complex128)
```

csdmpy.**plot**(*csdm_object*, *reverse_axis=None*, *range=None*, ***kwargs*)

A supplementary function for plotting basic 1D and 2D datasets only.

Parameters

- **csdm_object** – The CSDM object.
- **reverse_axis** – An ordered array of boolean specifying which dimensions will be displayed on a reverse axis.
- **kwargs** – Additional keyword arguments are used in matplotlib plotting functions. We implement the following matplotlib methods for the one and two-dimensional datasets.
  - The 1D{1} scalar dataset use the plt.plot() method.
  - The 1D{2} vector dataset use the plt.quiver() method.

- The 2D{1} scalar dataset use the plt.imshow() method if the two dimensions have a *linear* subtype. If any one of the dimension is *monotonic*, plt.NonUniformImage() method is used instead.
- The 2D{2} vector dataset use the plt.quiver() method.
- The 2D{3} pixel dataset use the plt.imshow(), assuming the pixel dataset as an RGB image.

Example

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

1.9.2 CSDM

class csdmpy.**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.
tags	List of tags attached to the dataset.
timestamp	Timestamp from when the file was last serialized.
geographic_coordinate	Geographic coordinate, if present, from where the file was last serialized.
dimensions	Tuple of the *Dimension* instances.
dependent_variables	Tuple of the *DependentVariable* instances.
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`	Add a new *Dimension* instance to the *CSDM* object.
`add_dependent_variable`	Add a new *DependentVariable* instance to the *CSDM* instance.
`to_dict`	Serialize the *CSDM* instance as a python dictionary.
`dumps`	Serialize the *CSDM* instance as a JSON data-exchange string.
`astype`	Return a copy of the CSDM object by converting the numeric type of each dependent variables components to the given value.
`save`	Serialize the *CSDM* instance as a JSON data-exchange file.
`copy`	Create a copy of the current CSDM instance.
`split`	Split the dependent-variables into view of individual csdm objects.

Numpy compatible attributes summary

`real`	Return a csdm object with only the real part of the dependent variable components.
`imag`	Return a csdm object with only the imaginary part of the dependent variable components.
`shape`	Return the count along each dimension of the csdm objects as a tuple.
`T`	Return a csdm object with a transpose of the dataset.

Numpy compatible method summary

`max`	Return a csdm object with the maximum dependent variable component along a given axis.
`min`	Return a csdm object with the minimum dependent variable component along a given axis.
`clip`	Clip the dependent variable components between the *min* and *max* values.
`conj`	Return a csdm object with the complex conjugate of all dependent variable components.
`round`	Return a csdm object by rounding the dependent variable components to the given *decimals*.
`sum`	Return a csdm object with the sum of the dependent variable components over a given *dimension=axis*.
`mean`	Return a csdm object with the mean of the dependent variable components over a given *dimension=axis*.
`var`	Return a csdm object with the variance of the dependent variable components over a given *dimension=axis*.

continues on next page

Table 5 – continued from previous page

`std`	Return a csdm object with the standard deviation of the dependent variable components over a given *dimension=axis*.
`prod`	Return a csdm object with the product of the dependent variable components over a given *dimension=axis*.

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.

tags
List of tags attached to the dataset.

timestamp
Timestamp from when the file was last serialized. This attribute is real only.

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. This attribute is read-only.

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 the *Dimension* instances.

dependent_variables
Tuple of the *DependentVariable* instances.

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.

Numpy compatible attributes documentation

real

Return a csdm object with only the real part of the dependent variable components.

imag

Return a csdm object with only the imaginary part of the dependent variable components.

shape

Return the count along each dimension of the csdm objects as a tuple.

T

Return a csdm object with a transpose of the dataset.

Methods documentation

add_dimension(**args*, ***kwargs*)

Add a new *Dimension* instance to the *CSDM* object.

There are several 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)
```

Using keyword as the arguments.

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

Using a Dimension class.

```
>>> var1 = Dimension(type = 'linear',
...                  increment = '5 G',
...                  count = 50,
...                  coordinates_offset = '-10 mT')
>>> datamodel.add_dimension(var1)
```

Using a subtype class.

```
>>> var2 = cp.LinearDimension(count = 50,
...                  increment = '5 G',
...                  coordinates_offset = '-10 mT')
>>> datamodel.add_dimension(var2)
```

From a numpy array.

```
>>> array = np.arange(50)
>>> dim = cp.as_dimension(array)
>>> datamodel.add_dimension(dim)
```

In the third and fourth example, the instances, `var1` and `var2` are added to the `datamodel` as a reference, *i.e.*, if the instance `var1` or `var2` is destroyed, the `datamodel` instance will become corrupt. As a recommendation, always pass a copy of the *Dimension* instance to the `add_dimension()` method.

add_dependent_variable(**args*, ***kwargs*)

Add a new *DependentVariable* instance to the *CSDM* instance.

There are again several 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.

to_dict (*update_timestamp=False*, *read_only=False*)

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.

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',
```

(continues on next page)

(continued from previous page)

```
'quantity_type': 'scalar', 'component_labels': ['response'],'components':
['AAAAABh5Fj9xeHM/cXhzPxh5Fj8yMQ0lGHkWv3F4c79xeHO/GHkWvw=='] }] }}
```

dumps (*update_timestamp=False*, *read_only=False*, *version='1.0'*, ***kwargs*)

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*, *indent=0*)

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 ith 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')
```

to_list()

Return the dimension coordinates and dependent variable components as a list of numpy arrays. For multiple dependent variables, the components of each dependent variable is appended in the order of the dependent variables.

For example,

- A 2D{1} will be packed as $[x_0, x_1, y_{0,0}]$
- A 2D{3} will be packed as $[x_0, x_1, y_{0,0}, y_{0,1}, y_{0,2}]$
- A 1D{1,2} will be packed as $[x_0, y_{0,0}, y_{1,0}, y_{1,1}]$

where x_i represents the ith dimension and $y_{i,j}$ represents the jth component of the ith dependent variable.

astype(*numeric_type*)

Return a copy of the CSDM object by converting the numeric type of each dependent variables components to the given value.

Parameters **numeric_type** – A numpy dtype or a string with a valid numeric type

Example

```
>>> data_32 = data_64.astype('float32')
```

copy()

Create a copy of the current CSDM instance.

Returns A CSDM instance.

Example

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

split()

Split the dependent-variables into view of individual csdm objects.

Returns A list of CSDM objects, each with one dependent variable. The objects are returned as a view.

Example

```
>>> # data contains two dependent variables
>>> d1, d2 = data.split()
```

transpose()

Return a transpose of the dependent variable data from the CSDM object.

fft(*axis=0*)

Perform a FFT along the given *dimension=axis*, for linear dimension assuming Nyquist-shannan relation.

Parameters **axis** – The index of the dimension along which the FFT is performed.

The FFT method uses the `complex_fft` attribute of the Dimension object to decide whether a forward or inverse Fourier transform is performed. If the value of the *complex_fft* is True, an inverse FFT is performed, otherwise a forward FFT.

For FFT process, this function is equivalent to performing

```
phase = np.exp(-2j * np.pi * coordinates_offset * reciprocal_coordinates)
x_fft = np.fft.fftshift(np.fft.fft(x)) * phase
```

over all components for every dependent variable.

Similarly, for inverse FFT process, this function is equivalent to performing

```
phase = np.exp(2j * np.pi * reciprocal_coordinates_offset * coordinates)
x = np.fft.ifft(np.fft.ifftshift(x_fft * phase))
```

over all components for every dependent variable.

Returns A CSDM object with the Fourier Transform data.

Numpy compatible method documentation

max (*axis=None*)

Return a csdm object with the maximum dependent variable component along a given axis.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is the sum over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a numpy array when dimension is None.

Example

```
>>> data2 = data.max()
```

min (*axis=None*)

Return a csdm object with the minimum dependent variable component along a given axis.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

clip (*min=None*, *max=None*)

Clip the dependent variable components between the *min* and *max* values.

Parameters

- **min** – The minimum clip value.
- **max** – The maximum clip value.

Returns A CSDM object with values clipped between min and max.

conj ()

Return a csdm object with the complex conjugate of all dependent variable components.

round (*decimals=0*)

Return a csdm object by rounding the dependent variable components to the given *decimals*.

sum (*axis=None*)
Return a csdm object with the sum of the dependent variable components over a given *dimension=axis*.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

mean (*axis=None*)
Return a csdm object with the mean of the dependent variable components over a given *dimension=axis*.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

var (*axis=None*)
Return a csdm object with the variance of the dependent variable components over a given *dimension=axis*.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

std (*axis=None*)
Return a csdm object with the standard deviation of the dependent variable components over a given *dimension=axis*.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the sum of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

prod (*axis=None*)
Return a csdm object with the product of the dependent variable components over a given *dimension=axis*.

Parameters **axis** – An integer or None or a tuple of *m* integers cooresponding to the index/indices of dimensions along which the product of the dependent variable components is performed. If None, the output is over all dimensions per dependent variable.

Returns A CSDM object with *m* dimensions removed, or a list when *axis* is None.

1.9.3 Dimension

LinearDimension

class csdmpy.**LinearDimension** (*count*, *increment*, *complex_fft=False*, ***kwargs*)
Bases: `csdmpy.dimensions.quantitative.BaseQuantitativeDimension`

LinearDimension class.

Generates an object representing a physical dimension whose coordinates are uniformly sampled along a grid dimension. See linearDimension_uml for details.

property absolute_coordinates
Return the absolute coordinates along the dimensions.

property axis_label
Return a formatted string for displaying label along the dimension axis.

property complex_fft
If True, orders the coordinates according to FFT output order.

property coordinates
Return the coordinates along the dimensions.

copy()
Return a copy of the object.

property count
Total number of points along the linear dimension.

property data_structure
Json serialized string describing the LinearDimension class instance.

property increment
Increment along the linear dimension.

reciprocal_coordinates()
Return reciprocal coordinates assuming Nyquist-shannan theorem.

reciprocal_increment()
Return reciprocal increment assuming Nyquist-shannan theorem.

to_dict()
Return the LinearDimension as a python dictionary.

property type
Return the type of the dimension.

MonotonicDimension

class csdmpy.**MonotonicDimension**(*coordinates*, ***kwargs*)
Bases: csdmpy.dimensions.quantitative.BaseQuantitativeDimension

A monotonic grid dimension.

Generates an object representing a physical dimension whose coordinates are monotonically sampled along a grid dimension. See monotonicDimension_uml for details.

property absolute_coordinates
Return the absolute coordinates along the dimensions.

property axis_label
Return a formatted string for displaying label along the dimension axis.

property coordinates
Return the coordinates along the dimensions.

property coordinates_offset
Value at index zero, c_k, along the dimension.

copy()
Return a copy of the object.

property count
Total number of points along the monotonic dimension.

property data_structure
Json serialized string describing the MonotonicDimension class instance.

to_dict()
Return the MonotonicDimension as a python dictionary.

property type
Return the type of the dimension.

LabeledDimension

class csdmpy.**LabeledDimension**(*labels*, *label=''*, *description=''*, *application={}*, ***kwargs*)
Bases: csdmpy.dimensions.base.BaseDimension

A labeled dimension.

Generates an object representing a non-physical dimension whose coordinates are labels. See labeledDimension_uml for details.

property axis_label
Return a formatted string for displaying label along the dimension axis.

property coordinates
Return the coordinates along the dimensions. This is an alias for labels.

copy()
Return a copy of the object.

property count
Total number of labels along the dimension.

property data_structure
Json serialized string describing the LabeledDimension class instance.

is_quantitative()
Return *True*, if the dimension is quantitative, otherwise *False*. :returns: A Boolean.

property labels
Return a list of labels along the dimension.

to_dict()
Return the LabeledDimension as a python dictionary.

property type
Return the type of the dimension.

class csdmpy.**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`	If true, the coordinates are the ordered as 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	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.
copy	Return a copy of the Dimension object.
reciprocal_coordinates	Return reciprocal coordinates assuming Nyquist-shannan theorem.
reciprocal_increment	Return reciprocal increment assuming Nyquist-shannan theorem.

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"
```

(continues on next page)

(continued from previous page)

```
...         }
... }
>>> 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

If true, the coordinates are the ordered as 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)
```

(continues on next page)

(continued from previous page)

```
[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",
  "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
```

copy()

Return a copy of the Dimension object.

reciprocal_coordinates()

Return reciprocal coordinates assuming Nyquist-shannan theorem.

reciprocal_increment()

Return reciprocal increment assuming Nyquist-shannan theorem.

1.9.4 DependentVariable

class csdmpy.**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 component 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`	Convert the unit of the dependent variable to the *unit*.
`to_dict`	Return DependentVariable object as a python dictionary.
`copy`	Return a copy of the DependentVariable object.

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

>>> 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"
...      }
```

(continues on next page)

(continued from previous page)

```
... }
>>> 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 component values from the dependent variable.

There are currently twelve *valid* numeric types in core scientific dataset model.

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

Besides, csdmpy also accepts any valid *type* object, such as int, float, np.complex64, as long as the type is consistent with the above twelve entries.

When assigning a valid value, this attribute updates the *dtype* of the Numpy array from the corresponding `components` attribute.

```
>>> 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]]

>>> y.numeric_type = float # python type object
>>> print(y.components[:,:5])
[[ 0.  1.  2.  3.  4.]
 [10. 11. 12. 13. 14.]
 [20. 21. 22. 23. 24.]]
```

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 kth *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:
...     print(e)
The shape of the `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"
    ],
    [
      "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]]}
```

copy()

Return a copy of the DependentVariable object.

1.9.5 Statistics

Methods Summary

integral	Evaluate the integral of the dependent variables over all dimensions.
mean	Evaluate the mean coordinate of a dependent variable along each dimension.
var	Evaluate the variance of the dependent variables along each dimension.
std	Evaluate the standard deviation of the dependent variables along each dimension.

Method Documentation

csdmpy.statistics.**integral**(*csdm*)

Evaluate the integral of the dependent variables over all dimensions.

Parameters **csdm** – A csdm object.

Returns A list of integrals corresponding to the list of the dependent variables. If only one dependent variable is present, return a quantity instead.

Example

```
>>> import csdmpy.statistics as stat
>>> x = np.arange(100) * 2 - 100.0
>>> gauss = np.exp(-((x - 5.) ** 2) / (2 * 4. ** 2))
>>> csdm = cp.as_csdm(gauss, unit='T')
>>> csdm.dimensions[0] = cp.as_dimension(x, unit="m")
>>> stat.integral(csdm)
<Quantity 10.0265131 m T>
```

csdmpy.statistics.**mean**(*csdm*)

Evaluate the mean coordinate of a dependent variable along each dimension.

Parameters **csdm** – A csdm object.

Returns A list of tuples, where each tuple represents the mean coordinates of the dependent variables. If only one dependent variable is present, return a tuple of coordinates instead.

Example

```
>>> stat.mean(csdm)
(<Quantity 5. m>,)
```

`csdmpy.statistics.var(csdm)`

Evaluate the variance of the dependent variables along each dimension.

Parameters **csdm** – A csdm object.

Returns A list of tuples, where each tuple is the variance along the dimensions of the dependent variables. If only one dependent variable is present, return a tuple instead.

Example

```
>>> stat.var(csdm)
(<Quantity 16. m2>,)
```

`csdmpy.statistics.std(csdm)`

Evaluate the standard deviation of the dependent variables along each dimension.

Parameters **csdm** – A csdm object.

Returns A list of tuples, where each tuple is the standard deviation along the dimensions of the dependent variables. If only one dependent variable is present, return a tuple instead.

Example

```
>>> stat.std(csdm)
(<Quantity 4. m>,)
```

1.9.6 Numpy methods

Supported NumPy functions

The csdm object supports the use of NumPy functions, as

```
>>> y = np.func(x)
```

where `x` and `y` are the csdm objects, and `func` is any one of the following functions. These functions apply to each component of the dependent variables from a given *csdm* object, *x*.

Trigonometric functions

The trigonometric functions apply to the components of the dependent variables from a csdm object.

Note: The components must be dimensionless quantities.

Table 11: A list of supported trigonometric functions.

Functions	Description
sin	Apply sine to the components of the dependent variables
cos	Apply cosine to the components of the dependent variables
tan	Apply tangent to the components of the dependent variables
arcsin	Apply inverse sine to the components of the dependent variables
arccos	Apply inverse cosine to the components of the dependent variables
arctan	Apply inverse tangent to the components of the dependent variables
sinh	Apply hyperbolic sine to the components of the dependent variables
cosh	Apply hyperbolic cosine to the components of the dependent variables
tanh	Apply hyperbolic tangent to the components of the dependent variables
arcsinh	Apply inverse hyperbolic sine to the components of the dependent variables
arccosh	Apply inverse hyperbolic cosine to the components of the dependent variables
arctanh	Apply inverse hyperbolic tangent to the components of the dependent variables

Mathematical operations

The following mathematical functions apply to the components of the dependent variables from a csdm object.

Note: The components must be dimensionless quantities.

Table 12: A list of supported mathematical functions.

Functions	Description
exp	Calculate the exponential of the components of the dependent variables.
expm1	Apply $e^x - 1$, where x are the components of the dependent variables.
exp2	Calculate 2^x, where x are the components of the dependent variables.
log	Calculate natural logarithm of the components of the dependent variables.
log1p	Calculate natural logarithm plus one on the components of the dependent variables.
log2	Calculate base-2 logarithm of the components of the dependent variables.
log10	Calculate base-10 logarithm of the components of the dependent variables.

The following mathematical functions apply to the components of the dependent variables from a csdm object irrespective of the components' dimensionality.

Table 13: Arithmetic operations

Functions	Description
reciprocal	Return element-wise reciprocal.
positive	Return element-wise numerical positive.
negative	Return element-wise numerical negative.

Table 14: Miscellaneous

Functions	Description
sqrt	Return element-wise non-negative square-root.
cbrt	Return element-wise cube-root.
square	Return element-wise square.
absolute	Return element-wise absolute value.
fabs	Return element-wise absolute value.
sign	Return element-wise sign of the values.

Table 15: Handling complex numbers

Functions	Description
angle	Return element-wise angle of a complex value.
real	Return element-wise real part of a complex value.
imag	Return element-wise imaginary part of a complex value.å
conj	Return element-wise conjugate.
conjugate	Return element-wise conjugate.

Table 16: Sums, products, differences

Functions	Description
prod	Return the product of the components of a dependent variable along a dimension.
sum	Return the sum of the components of a dependent variable along a dimension.

Table 17: Rounding

Functions	Description
rint	Round elements to the nearest integer.
around	Round elements to the given number of decimals.
round	Round elements to the given number of decimals.

Other functions

- min
- max
- mean
- var
- std

Dimension specific Apodization methods

The following methods of form

$$y = f(ax),$$

where a is the function argument, and x are the coordinates along the dimension, apodize the components of the dependent variables along the respective dimensions. The dimensionality of a must be the reciprocal of that of x. The resulting CSDM object has the same number of dimensions as the original object.

Method Summary

`sin`(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\sin(ax)$.
`cos`(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\cos(ax)$.
`tan`(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\tan(ax)$.

continues on next page

Table 18 – continued from previous page

arcsin(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\arcsin(ax)$.
arccos(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\arccos(ax)$.
arctan(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\arctan(ax)$.
exp(csdm, arg[, dimension])	Apodize the components along the *dimension* with $\exp(ax)$.

Method Documentation

`csdmpy.apodize.`**`sin`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\sin(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the sine of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`cos`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\cos(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the cosine of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`tan`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\tan(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the tangent of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`arcsin`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\arcsin(ax)$.

Parameters

- **csdm** – A CSDM object.

- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the inverse sine of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`arccos`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\arccos(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the inverse cosine of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`arctan`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\arctan(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the inverse tangent of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

`csdmpy.apodize.`**`exp`**(*csdm*, *arg*, *dimension=0*)

Apodize the components along the *dimension* with $\exp(ax)$.

Parameters

- **csdm** – A CSDM object.
- **arg** – String or Quantity object. The function argument a.
- **dimension** – An integer or tuple of *m* integers cooresponding to the index/indices of the dimensions along which the exp of the dependent variable components is performed.

Returns A CSDM object with *d-m* dimensions, where *d* is the total number of dimensions from the original *csdm* object.

CHAPTER

TWO

CITATION

CHAPTER

THREE

CHECK OUT THE MEDIA COVERAGE

INDEX

A

C

D

U

V