



**HAL**  
open science

# A Unified Framework for Manipulating N-dimensional Astronomical Data and Coordinate Transformations in Python: The NDCube 2 and Astropy APE-14 World Coordinate System APIs

Daniel Ryan, Stuart Mumford, Will Barnes, Ankit Kumar Baruah, Adwait Bhope, Éric Buchlin, Nabil Freij, Adam Ginsburg, Laura Hayes, Derek Homeier, et al.

## ► To cite this version:

Daniel Ryan, Stuart Mumford, Will Barnes, Ankit Kumar Baruah, Adwait Bhope, et al.. A Unified Framework for Manipulating N-dimensional Astronomical Data and Coordinate Transformations in Python: The NDCube 2 and Astropy APE-14 World Coordinate System APIs. *The Astrophysical Journal*, 2023, 956 (1), pp.44. 10.3847/1538-4357/ace0bd . hal-04252174

**HAL Id: hal-04252174**

**<https://hal.science/hal-04252174v1>**

Submitted on 26 Oct 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.



















L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License



# A Unified Framework for Manipulating N-dimensional Astronomical Data and Coordinate Transformations in Python: The NDCube 2 and Astropy APE-14 World Coordinate System APIs

Daniel F. Ryan<sup>1,2,19</sup> , Stuart Mumford<sup>3,19</sup> , Will T. Barnes<sup>2,4</sup> , Ankit Kumar Baruah<sup>5</sup>, Adwait Bhope<sup>6</sup> , Éric Buchlin<sup>7</sup> , Nabil Freij<sup>8,9</sup> , Adam Ginsburg<sup>10</sup> , Laura A. Hayes<sup>11</sup> , Derek Homeier<sup>3</sup> , J. Marcus Hughes<sup>12</sup> , Chris Lowder<sup>12</sup> , Richard O'Steen<sup>13</sup> , Baptiste Pellorce<sup>14,15</sup>, Thomas Robitaille<sup>3</sup> , Yash Sharma<sup>16</sup>, David Stansby<sup>17</sup> , Albert Y. Shih<sup>4</sup> , Erik Tollerud<sup>13</sup> , Micah J. Weberg<sup>18</sup> , and Matthew J. West<sup>12</sup> 

<sup>1</sup> University of Applied Sciences Northwest Switzerland, Bahnhofstrasse 6, 5210 Windisch, Switzerland; [ryand5@tcd.ie](mailto:ryand5@tcd.ie)

<sup>2</sup> American University, 4400 Massachusetts Avenue NW, Washington, DC 20016, USA

<sup>3</sup> Aperio Software Ltd., UK

<sup>4</sup> NASA Goddard Space Flight Center, 8800 Greenbelt Road, Greenbelt, MD 20771, USA

<sup>5</sup> Workato GmbH, Westendstrasse 28, D-60325 Frankfurt/Main, Germany

<sup>6</sup> Uptycs India Pvt. Ltd., Aundh IT Park, Pune, MH 411020, India

<sup>7</sup> Université Paris-Saclay, CNRS, Institut d'Astrophysique Spatiale, F-91405, Orsay, France

<sup>8</sup> Lockheed Martin Solar and Astrophysics Laboratory, Palo Alto, CA 94304, USA

<sup>9</sup> Bay Area Environmental Research Institute, Moffett Field, CA 94035, USA

<sup>10</sup> Department of Astronomy, University of Florida, Bryant Space Science Center, Gainesville, FL 32611, USA

<sup>11</sup> European Space Agency, ESTEC, Keplerlaan 1—2201 AZ, Noordwijk, The Netherlands

<sup>12</sup> Southwest Research Institute, 1050 Walnut Street, Suite 300, Boulder, CO 80302, USA

<sup>13</sup> Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218, USA

<sup>14</sup> Claude Bernard Lyon 1 University, 43 Bd du 11 Novembre 1918, F-69100 Villeurbanne, France

<sup>15</sup> Institute of Theoretical Astrophysics, Sem Sælands vei 13, NO-0371 Oslo, Norway

<sup>16</sup> Meta Platforms Inc., 10 Brock Street, Regents Place, London, NW1 3FG, UK

<sup>17</sup> Advanced Research Research Computing Centre, University College London, Gower Street, London, WC1E 6BT, UK

<sup>18</sup> George Mason University, 4400 University Drive, Fairfax, VA 22030, USA

Received 2023 March 7; revised 2023 June 9; accepted 2023 June 20; published 2023 October 6

## Abstract

The NDCube 2 API is a Python application programming interface (API) for storing and manipulating N-dimensional coordinate-aware astronomical data. While there are Python packages for handling astronomical data and coordinate transformations separately and for handling specific combinations of dimensions and transformations, none provide a unified and agnostic way of handling them simultaneously. This leads to a proliferation of different APIs for conducting the same analysis tasks on similar types of observations and introduces technical barriers between multi-instrument studies and cross-community collaboration. In this paper, we outline how the NDCube 2 API and its implementation in the open-source, community-developed `ndcube` package, together with the `AstroPy` WCS API, help to solve this problem. We discuss the guiding principles underpinning the API design and provide examples of how it is already being used to serve broad sections of the astronomy community, including agency-funded missions. The aim of this paper is to help users better understand the purpose and potential of the NDCube 2 API and `ndcube` package and hence how to more effectively deploy them in scientific analyses and software development.

*Unified Astronomy Thesaurus concepts:* [Astronomy data analysis \(1858\)](#); [Astronomical coordinate systems \(82\)](#); [Astronomy software \(1855\)](#); [Open source software \(1866\)](#); [Distributed computing \(1971\)](#); [GPU computing \(1969\)](#)

Software reviewed by the [Journal of Open Source Software](#) 

## 1. Introduction

The analysis of N-dimensional (ND) data and the physical coordinates their dimensions represent is a fundamental pillar of astronomy. 2D frequency–time radio spectrograms, 3D time–space–space ultraviolet (UV) image stacks, and 4D time–space–space–stokes cubes produced by polarimetric imagers are just three examples. In computer analysis, these data are stored and manipulated in ND arrays. The value in each array element represents a measurement of a physical property (e.g., intensity), the index of the element represents the location

in the universe being sampled, and the array axes represent the physical types that define that location (e.g., time, space, wavelength, etc.).

In astronomy, the relationship between array indices and real world coordinates is typically represented via the World Coordinate System (WCS). WCS is a broad framework with multiple implementations. The most common is Flexible Image Transport System (FITS)-WCS (Calabretta & Greisen 2002; Greisen & Calabretta 2002; Greisen et al. 2006; Rots et al. 2015) designed for use in FITS files (Wells et al. 1981; Hanisch et al. 2001; Pence et al. 2010), but others include generalized WCS designed for the James Webb Space Telescope (JWST) and LSST-WCS designed for the Legacy Survey of Space and Time (Ivezić et al. 2019). Due to the importance of WCS, the `AstroPy` project (Astropy Collaboration et al. 2013, 2018) has developed tools in the Python programming language to store, inspect, and execute WCS transformations, e.g., the `gWCS`

<sup>19</sup> Daniel F. Ryan and Stuart Mumford contributed equally to this work.

package (Dencheva et al. 2023) and the WCS module in `astropy` package.<sup>19</sup> However, the application programming interfaces (APIs) of these tools differ based on the WCS implementation for which they were designed. This causes WCS-based user tools and pipelines to become implementation-specific and limits their broader utility.

There are mature Python packages that support ND array manipulation, such as `numpy` (Harris et al. 2020) and `dask`.<sup>20</sup> However, neither they nor the above-mentioned WCS tools are suited to treating data and WCS coordinate transformations in a combined way. The closest preexisting candidate is `xarray` (Hoyer & Hamman 2017). However, `xarray` has been developed for the requirements and conventions of the geosciences, which, although similar to those of astronomy in concept, are sufficiently different in construction to cause significant friction. Crucially, `xarray` does not support WCS coordinate transformations but rather stores coordinates as lookup tables. The tools that do support a WCS-based coordinate-aware data analysis, such as the SunPy (Mumford et al. 2020) Map class for 2D images of the Sun, tend to have APIs specific to particular combinations of dimensions, physical types, coordinate systems, and WCS implementations. This limits their broader utility and makes the combined analysis of different types of data more difficult. It also inhibits collaboration by putting technical barriers between subfields of astronomy.

In this paper, we outline how the NDCube 2 API addresses the above challenges. It provides a generalized WCS-based coordinate-aware framework for ND Python data analysis. It is agnostic to the number of dimensions, combination of physical coordinate types, and the underlying WCS implementation. Note that the goal of this paper is not to provide detailed user instructions or an exhaustive list of functionalities offered by the NDCube 2 API or the `ndcube` package. For this, we refer readers to API’s defining document (Mumford & Ryan 2020) in addition to the `ndcube` package publication in JOSS (Ryan et al. 2023a) and the `ndcube` documentation.<sup>21</sup> Instead, this paper aims to reveal the philosophies and capabilities of these tools. It is hoped this will help users and developers understand the role that the NDCube 2 API plays in the scientific Python ecosystem and how it can be applied to their use-cases. In Section 2, we discuss the benefits of WCS to astronomy over simple coordinate lookup tables. We then outline one of the pillars of the NDCube 2 API, the AstroPy WCS API, a standardized API that can be wrapped around specific WCS implementations. In Section 3, we outline the NDCube 2 API itself and highlight some challenges and philosophies that led to specific API designs. In Section 4, we discuss the `ndcube` package,<sup>22</sup> which implements the NDCube 2 API as well as some additional tools. In Section 5, we give an example of how the NDCube class can be used to analyze real 4D observations with spatial, spectral, and temporal axes. In Section 6, we demonstrate the usefulness of the NDCube 2 API by highlighting its deployment by packages that serve broad subsections of the astronomy community as well as specific agency-funded missions. In Section 7, we discuss the the future of the

`ndcube` package before finally providing a summary and conclusion in Section 8. Note that in the Appendices we clarify some terminology used throughout this paper. Readers not intimately familiar with Python, Astropy, and WCS are encouraged to read this section before continuing as it will make the discussion that follows easier to understand.

## 2. The World Coordinate System

### 2.1. Why WCS?

The WCS, is a framework for representing astronomical transformations between *pixel* indices<sup>23</sup> and real world coordinates. Although pervasive throughout astronomy, WCS can seem esoteric when first encountered. Because of this, some scientists prefer to convert the WCS information to lookup tables giving the coordinate values of each element in their data array. This enables subsequent analysis to be done in a simpler array-based way. While this approach is valid, it has drawbacks.

WCS is a functional framework meaning it can preserve the underlying mathematics of the transformations and execute them on-demand. This makes WCS well-suited to describing observations that smoothly sample a contiguous region of a coordinate space, e.g., an image of the sky. WCS transformations can be applied continuously throughout and beyond the data grid. They can be evaluated for subregions within pixels by using noninteger pixel indices, or beyond the field of view by using indices beyond the extent of the array. This makes it easier to combine and compare observations with different plate scales and fields of view. Finally, functional coordinate transformations can be highly memory-efficient as they can often be fully described by only a few parameters. This contrasts with coordinate lookup tables that give the discrete values of the coordinates for each element in the data array, like those used by `xarray`.

Although coordinate lookup tables are conceptually simpler, their memory inefficiency scales with the size of the data. Moreover, they do not preserve the mathematics of the transformations. Therefore, resampling the data requires the lookup table to be interpolated, which may only approximate the true transformations. However, coordinate lookup tables are necessary when there are discontinuities in the transformations, e.g., irregular time steps. In such cases, the WCS framework can still be used by making the functional transformations refer to or interpolate lookup tables. However, if all your coordinate transformations are discontinuous, the WCS and NDCube 2 API frameworks provide fewer advantages over lookup tables and `xarray`, except when users want to employ tools that depend on those frameworks. Nonetheless, in the cases involving continuous coordinate transformations, WCS provides greater flexibility and utility over lookup tables.

### 2.2. The Astropy WCS API

There are multiple implementations of the WCS framework each with a different, incompatible API, which can significantly hinder interoperability and collaboration. The AstroPy WCS API (often referred to as the APE 14 API due of its origin in the 14th Astropy Proposal for Enhancement; Robitaille et al.

<sup>19</sup> <https://gwcs.readthedocs.io/en/stable/>

<sup>20</sup> <https://dask.org>

<sup>21</sup> <https://docs.sunpy.org/projects/ndcube/en/stable/>

<sup>22</sup> Version (1) of the `ndcube` package predates the Astropy WCS and the NDCube 2 APIs and is not compatible with them. Therefore all discussion in this paper will refer to version (2) and later.

<sup>23</sup> Although the term *pixel indices* is used by WCS for historical reasons, it can refer to the indices of any data array irrespective of whether the observing instrument uses pixels or not. See Appendix A.1 for further details and the distinction between *pixel* and *array* indices.

2018) solves this problem by defining a standardized API for inspecting and executing WCS transformations. Instead of simply proliferating the number of incompatible WCS APIs, the AstroPy WCS API allows different implementations to hook into it, effectively acting as a translation layer from a standardized user-facing API to any underlying WCS implementation. The creation of the AstroPy WCS API was partly motivated by the need for a compatible interface to both the FITS-WCS (Calabretta 2011) and generalized WCS (Dencheva & Greenfield 2019) implementations. Its success in that role has demonstrated the API’s ability to encourage and facilitate interoperability by liberating users and developers from caring about the specific underlying implementation of their WCS object.

The AstroPy WCS API is composed of two tiers: low-level and high-level. Both provide methods for executing the transformations, but the low-level API also provides access to important information about the transformations such as the axis correlation matrix and the physical types of the world axes. The low-level API accepts and returns basic Python objects like scalars, arrays, and strings, making it simpler to implement. As such, the methods that execute the transformations tend to return raw output, i.e., values without important context information such as units, coordinate frame, epoch, etc. Instead, this information is available elsewhere in the low-level API. While this more closely maps to how the underlying information is stored, most end users would be better served if the raw output and relevant context information were combined. This is the role of the high-level API, which accepts and returns high-level objects—e.g., `astropy Time`, `SkyCoord`, etc.—based on the information available in the low-level API. To facilitate development, AstroPy provides an object that can wrap any low-level-API-compliant object and expose the high-level API. This means developers need only define the translation between their underlying WCS implementation and the simpler low-level API, thus reducing development duplication. Moreover, the AstroPy WCS API is package-independent,<sup>24</sup> thus facilitating interoperability between packages outside of AstroPy, even non-Python packages.

The original proposed AstroPy WCS API and the motivation behind it are discussed in Robitaille et al. (2018), and its official definition can be found in the `BaseLowLevelWCS` and `BaseHighLevelWCS` classes in the `astropy` Codebase (Astropy Development Team 2023). We therefore refer readers to the above references for more detail. But the above summary reveals that the two-tiered AstroPy WCS API provides an intuitive and dependable WCS API for users and developers, and encourages interoperability between WCS implementations. Such an API is a crucial component of a generalized API for analyzing WCS-based, coordinate-aware, ND data, namely, the NDCube 2 API.

### 3. The NDCube 2 API

The NDCube 2 API is a standardized framework for inspecting and manipulating coordinate-aware ND data. It is defined by Mumford & Ryan (2020) and is split over three abstract base classes (ABCs): `NDCubeABC`, `ExtraCoordsABC`, and `GlobalCoordsABC`. It relies on two foundational assumptions. First, the data are stored in a single array object, and second, the primary set of coordinates is stored in an object

that complies with the AstroPy WCS API (for cases where the data are split over multiple arrays, see discussions of the `NDCubeSequence` and `NDCollection` classes in Sections 4.2 and 4.3). From these assumptions flow a set of functionalities that are independent of the number of dimensions and physical coordinate types they represent. The resulting API allows users and developers to intuitively interact with coordinate-aware ND data analogous to how they use arrays to interact with coordinate-agnostic ND data.

The NDCube 2 API is agnostic to the type of underlying array infrastructure. Therefore, while it can be used with `numpy` arrays, the NDCube 2 API is valid for `cupy` arrays for GPU-enabled computation (Okuta et al. 2017) as well as `dask` arrays for parallel and out-of-core processing. In the case of `dask` arrays, this means that the arrays that exceed or are comparable to a single computer’s RAM can be inspected and sliced using the NDCube 2 API without actually bringing that array into memory. Furthermore, coordinate-aware computation can be performed on the underlying array in a similar manner and can be parallelized across multiple cores or even multiple machines using `dask`.

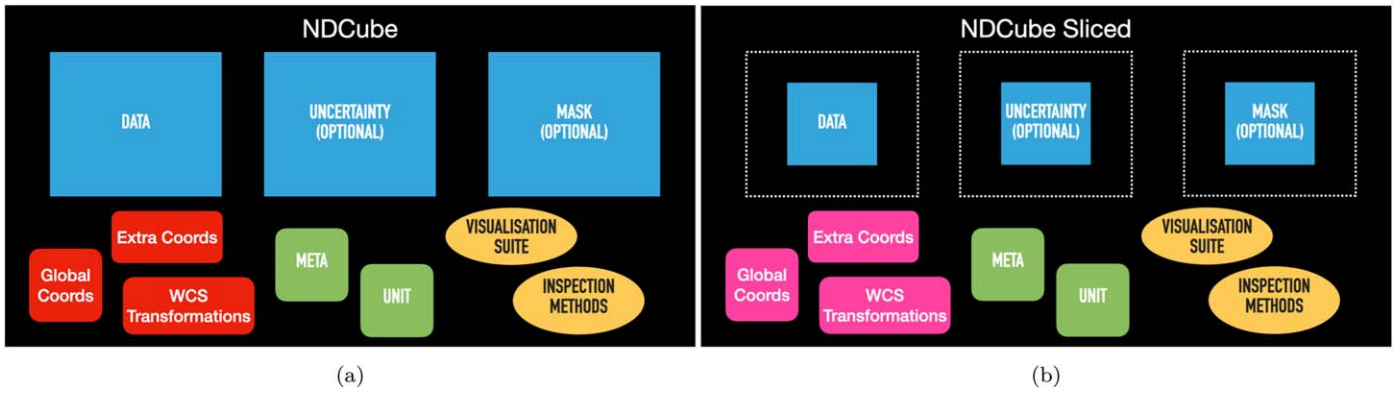
The versatility and standardization of the NDCube 2 API facilitates and encourages interoperability between different analysis tools. Users can perform the same tasks in the same ways whether they represent different physical types—e.g., images versus spectra—or rely on different underlying infrastructure—e.g., `numpy` versus `dask` arrays. This streamlines the user experience and provides a dependable foundation upon which developers can build more specialized tools.

#### 3.1. *ExtraCoordsABC* and *GlobalCoordsABC*

As well as `NDCubeABC` (Section 3.2), the NDCube 2 API includes two supplemental coordinate classes, `ExtraCoordsABC` and `GlobalCoordsABC`. `ExtraCoordsABC` allows users to store an alternative or complimentary set of coordinates to the primary WCS. We shall see in Section 3.2 that the `NDCubeABC` allows users to utilize the primary WCS or `ExtraCoords` transformations alternately so long as transformations for all array axes are contained in the `ExtraCoords` object. This is a requirement of the WCS framework. However, `ExtraCoordsABC` also supports transformations for as few array axes as desired in either functional or lookup table form. This means `ExtraCoordsABC` can store partial sets of discontinuous coordinate transformations that are supplemental, rather than an alternative, to the primary WCS object. To clarify this, consider a stack of images at different wavelengths where each wavelength image is taken at a different time. The array axes have physical types of space, space, and wavelength–time. However, say the primary WCS only describes the celestial and spectral world axes. This could be because the time intervals are irregular, and the underlying WCS implementation is FITS-WCS, which is not well-suited to storing discontinuous transformations. Such cases are common in solar physics. Including the timestamps in the primary WCS would require the user to build a new WCS object from scratch and may require them to abandon the FITS-WCS standard for one better suited to tabular transformations. This can be a difficult and complicated task, even for expert users. An easier solution would be to store the timestamps in the `ExtraCoords` object. Thus, because the `ExtraCoords` object stores transformations for only a subset (one) of the array axes, it is supplemental

<sup>24</sup> `astropy` implements the WCS API for both FITS-WCS and generalized WCS, demonstrating that the standard is implementable and useful.





**Figure 1.** (a) Components of a basic NDCube 2 API-compliant object. Array-based components are blue, coordinate components red, metadata components green, and inspection, analysis, and visualization methods are yellow. (b) The effect of slicing an NDCubeBase instance by array indices via the standard Python slicing API or by world coordinates via the `.crop` method. The array components have been cropped in accordance with the input slice item, and the coordinate objects have been altered so that array elements correspond to the same world coordinates as before. This is achieved with a single line of code making extracting regions of interest easier and less prone to error.

to the primary WCS object. However, if it contained transformations for all array axes, it could be supplemental or an alternative to the primary WCS object.

The `GlobalCoordsABC` serves a subtly different purpose. It facilitates scalar coordinates that apply to the data cube as a whole rather than any subset of its axes. Consider a 3D image-time cube from which we extract a single 2D image. The image is now associated with a scalar timestamp that is still a valid coordinate as it indicates where in the universe (in time) the image corresponds. However, the WCS framework cannot support it as it no longer corresponds to an axis of the data array. Therefore, the `GlobalCoordsABC` was developed to store and track such information. Slicing is an obvious case where global coordinates can be generated and tracked. Hence, the slicing infrastructure of the `NDCubeBase` class in the `ndcube` package, which implements the NDCube 2 API, automatically adds sliced scalar coordinates to its associated `GlobalCoords` class as required (Section 3.2.2). However, because users may need to manipulate global coordinates outside of slicing operations, an `NDCubeBase` instance is always expected to have an associated `GlobalCoords`—even if empty—to which users can manually add and remove global coordinates as needed.

### 3.2. NDCubeABC

#### 3.2.1. Structure

`NDCubeABC` is the primary class of the NDCube 2 API. It inherits from `astropy.nddata.NDDataBase` and so stores its core information in the same way. Figure 1(a) shows a schematic of the `NDCubeABC` class. The blue squares represent the array-based properties, `.data`, `.uncertainty`, `.mask`. These hold the data array, the uncertainty of each data element, and a boolean array denoting which elements are reliable (i.e., a mask array). The `.uncertainty` and `.mask` attributes must have the same dimensions as `.data` unless they are set to `None` (`.mask` can also be set to a boolean if all elements have the same mask value). Metadata attributes are shown in green. `.meta` contains general metadata such as the instrument used to make the observations while `.unit` gives the unit of `.data`. Coordinate attributes are shown in red. `.wcs` stores the primary WCS object, `.extra_coords` holds an `ExtraCoordsABC`-compliant object, and `.global_coords` holds a `GlobalCoordsABC`-

compliant object. The only components required to instantiate an NDCube 2 API-compliant object are a data array and an `AstroPy`-WCS-API-compliant object. By default, `.extra_coords` and `.global_coords` are empty while other attributes can be set to `None`.

`NDCubeABC` provides convenience methods that make exploring and analyzing coordinate-aware data more powerful and straightforward. The guiding philosophy behind the scope of the NDCube 2 API is that all functionalities should depend on information from multiple components of the `NDCubeABC` and be independent of the number of dimensions and the physical types they represent. This is demonstrated by the `.array_axis_physical_types` property, which returns the physical types associated with each array axis. This is different from the `world_axis_physical_types` property on the low-level `AstroPy` WCS API, which returns the single physical type associated with each world axis. By contrast, `.array_axis_physical_types` may return multiple physical types per array axis, and the same world axis may be associated with multiple array axes. This is because multiple world axes can be associated with a single pixel axis via the axis correlation matrix and vice versa (see Appendix A.1 for explanations of the differences between array, pixel, and world axes and Appendix A.2 for discussion of the axis correlation matrix). Moreover, because coordinate transformations can be stored in the `.wcs` and `.extra_coords` attributes, determining the complete set of physical types associated with each array axis requires knowledge of both components and so is within the scope of the NDCube 2 API.

Another example is the `.axis_world_coords` method, which returns the coordinate values for every element in the data grid. While the coordinates for user-defined array indices can be calculated directly via the primary WCS or `ExtraCoordsABC` objects, `.axis_world_coords` derives the grid of array indices for the whole data array and passes them to the relevant transformation object. It then returns the world coordinates in array order, so they can be easily compared to the data array. This liberates users from a confusing and tedious process, thus enhancing their productivity. That said, if the coordinates of only a few array elements are required, directly executing the transformations via the WCS object can be more efficient. `NDCubeABC` provides a very similar method called `.axis_world_coords_values`, which differs only in the

types of objects returned. Whereas `.axis_world_coords` returns high-level objects like `astropy.time.Time`, `astropy.coordinates.SkyCoord`, etc., `.axis_world_coords_values` returns arrays of raw values without context information like the coordinate frame. This can be more efficient and useful for developers, but most end users will be better served by `.axis_world_coords`. This is analogous to the division of low-level and high-level APIs within the Astropy WCS API.

The `.axis_world_coords/.axis_world_coords_values` methods also demonstrate how the NDCube 2 API facilitates the interplay between the `.wcs` and `.extra_coords` objects. The methods enable users to select which set of transformations to use by setting the `wcs=` kwarg to the desired transformation object. This option is available for all NDCube 2 API methods where users might want to alternate between the primary WCS and `ExtraCoords` transformations. Users can also set the `wcs=` kwarg to a WCS object that combines the primary WCS and the `ExtraCoords`, provided by the `NDCubeABC.combined_wcs` property.

For a full definition of all methods and properties available via the NDCube 2 API, see Mumford & Ryan (2020). In the following subsection, we will demonstrate the power of the NDCube 2 API by focusing on a specific analysis task: extracting regions of interest.

### 3.2.2. Slicing and Cropping: Extracting Regions of Interest

One of the most powerful consequences of combining data and coordinates in a single NDCube 2 API-compliant object is the ability to extract regions of interest. Let us assume without loss of generality that Figure 1(a) represents a 3D time-image cube. We can extract a region of interest over a certain time period by simply applying the standard Python slicing API,

```
1. >>> sliced_cube = my_cube[z1:z2, y1:y2, x1:x2]
```

where `z1`, `z2`, `y1`, `y2`, `x1`, and `x2` are integers representing indices along the array axes. To help remind themselves which array axes correspond to which world axes, users can call the `.array_axis_physical_types` that returns the physical coordinate type(s) associated with each array axis, as discussed in Section 3.2.1. For the above 3D time-image cube the example, the result might be something like

```
1. >>> my_cube.array_axis_physical_types
2. [('time'), ('pos.eq.ra', 'pos.eq.dec'), ('pos.eq.ra', 'pos.eq.dec)']
```

Each tuple corresponds to an array axis and the string(s) within gives the associated physical type(s). From this, we can see that the first array axis corresponds to time, while the second and third correspond to the spatial coordinate types of right ascension (R.A.) and declination (decl.). This API immediately shows that the first axis is independent as its physical coordinate types are not associated with any other axes. Meanwhile, the second and third array axes are dependent as they both correspond to both spatial coordinate types. This API also identifies the spatial coordinate system, namely, R.A. and decl. A different spatial coordinate system, e.g., helioprojective, would cause `.array_axis_physical_types` to output different spatial coordinate types.

Figure 1(b) shows the result of the slicing operation demonstrated above. The data, uncertainty, and mask arrays have been cropped in accordance with the slice item. The

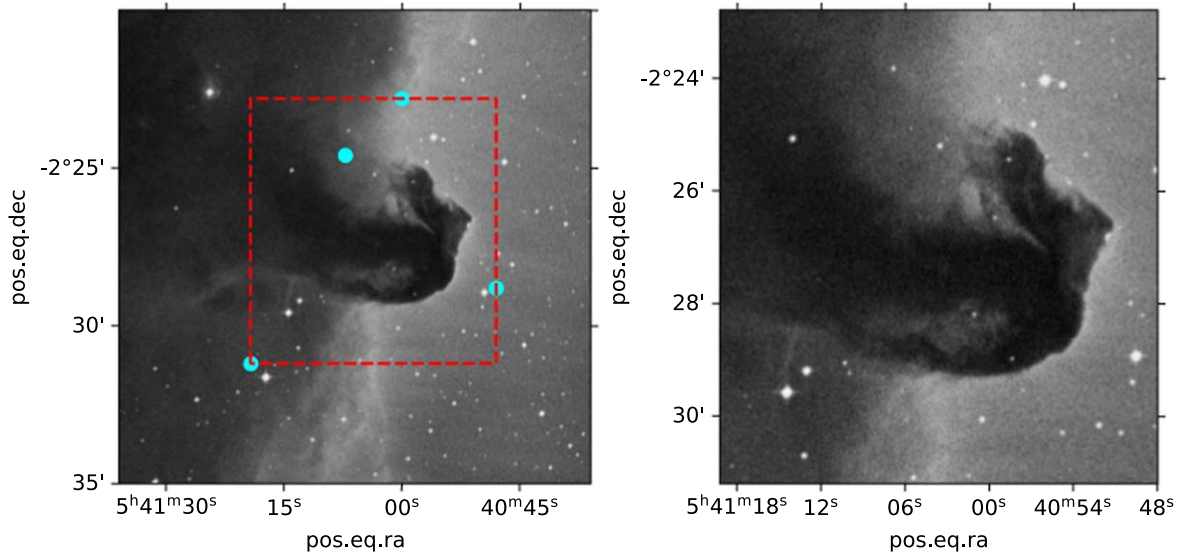
coordinate objects are typically functional rather than array-based and so are not cropped. Instead, the WCS is modified via Astropy's `SlicedLowLevelWCS` wrapper class while the `ExtraCoords` instance has been altered by its own internal slicing infrastructure. Thus, the same array elements correspond to the same real world coordinates, even though their indices have been changed by the slicing process. If the data dimensionality is reduced by the slicing, the coordinate values corresponding to relevant locations along the dropped axes are added to the `GlobalCoords` instance. For example, if we slice out the fourth 2D image in the image-time cube, the fourth timestamp will be added to `.global_coords`. If each of the `NDCubeABC` components were stored separately, the arrays would have to be sliced individually, and the coordinate objects would have to be altered to reflect the new shape of the arrays. This can be a time-consuming and error-prone process. The `NDCubeABC` slicing API makes this simple and reliable.

A region of interest can also be extracted using world coordinates via the `NDCubeABC.crop` API. This method takes tuples of high-level scalar astropy coordinate objects that describe individual locations within the cube. An example of this is shown by the cyan points in the left panel of Figure 2, which represents an image of the sky stored in a 2D `NDCubeABC` instance. The method crops the instance to the smallest bounding box in array-index space containing all the locations (red dashed box, left panel Figure 2). This ensures a cube-like data shape is maintained, but does mean that some array elements outside the region of interest may be kept if the pixel and world coordinate grids are not aligned. Also note that redundant locations can be provided that do not alter the bounding box, as shown by the cyan point inside the red dashed bounding box in the left panel of Figure 2. The right panel of Figure 2 shows the resultant cropped image, which was achieved via the following command:

```
1. >>> my_cube.crop((celestial1,), (celestial2,), (celestial3,), (celestial4,))
```

Each `celestial` variable is an `astropy.coordinates.SkyCoord` object. Note that, because Figure 2 shows a 2D celestial `NDCube`, each location is uniquely described by a single `SkyCoord`. However, higher dimensional cubes may also require an instance of `astropy.time.Time`, `astropy.coordinates.SpectralCoord`, etc. Note that there is also a `.crop_by_values` version of the method that accepts low-level coordinate objects, analogous to `.axis_world_coords_values`. Also akin to `.axis_world_coords/.axis_world_coords_values`, `.crop` and `.crop_by_values` enable users to use the primary WCS object, `ExtraCoords`, or a WCS combining them via a `wcs=` kwarg.

The `crop` API is simple and intuitive in light of how flexible and powerful it is. However, it may appear cumbersome when applied to a simple data type, e.g., a 2D image. This highlights the trade-off between generality and simplicity when designing the NDCube 2 API. For example, we initially tried to develop an API that did not require users to provide inputs for real world coordinate types if the region's limits were not defined by them. We hoped that missing objects could be inferred by comparing the input objects with the coordinate types listed in the WCS object. This would have meant that, if, for example, a user wanted to crop in space but not in time, they would only have to provide `SkyCoord` objects. However, it was discovered that



**Figure 2.** Demonstration of cropping a 2D celestial NDCube via the `.crop` method. Left: the uncropped image. The cyan dots represent the locations input to `.crop` while the dashed red line represents the minimal array-index-space bounding box. The point within the bounding box is redundant. Right: the resultant cropped image corresponding to the bounding box.

this cannot be implemented in a generalized way without prohibitive complexity. Hence, the `crop` API requires an object to be provided in place of each coordinate type for each input point. That said, if a user’s region of interest is independent of a coordinate type, e.g., time, `None` can be provided in place of the `Time` objects. This saves the user pointlessly working out the extremities of the time range of their data. Another example was our attempt to design the API to accept one instance of each coordinate type giving the lower and upper limits of the bounding box in each world axis. However, it was quickly realized that a region of interest can only be uniquely defined in this way when the pixel grid is aligned with the world coordinate grid, or the axis correlation matrix is diagonal.

The resulting complexities in the API are therefore unavoidable consequences of the NDCube 2 API’s guiding principle that all functionalities must be independent of the number of dimensions and combination of physical types. However, if developers are working on a less generalized class, they can adopt the NDCube 2 API and supplement it with additional, more user-friendly APIs. In such cases, the NDCube 2 API can still be leveraged. For example, a new `crop` method with an API better suited to a specific number of axes and combination of physical types only needs to translate between the generalized `NDCubeABC.crop` API and the more specialized one. The cropping itself can then still be done by the underlying `NDCubeABC.crop` method. Furthermore, by supporting the NDCube 2 API as well as more specialized ones, developers can simultaneously make their objects more user-friendly and compatible with tools from other packages that rely on the NDCube 2 API. This encourages interoperability and greater productivity of users and developers.

#### 4. The `ndcube` Package

The `ndcube` package (Ryan et al. 2023a; `ndcube` Development Team 2023) is an open-source Python package that serves three purposes.<sup>25</sup> First, it formalizes the NDCube 2 API via ABCs in its code-base. This provides a simple way for developers to

ensure their classes adhere to the NDCube 2 API by having their classes inherit the ABCs. It also enables users to check whether an object is NDCube 2 API-compliant by checking whether it is an instance of one of the ABCs, e.g.,

```
1. >>> from ndcube import NDCubeABC
2. >>> if isinstance(my_obj, NDCubeABC):
3. >>> ... print('my_obj is NDCube-2-API-compliant!')
```

Second, the `ndcube` package implements the NDCube 2 API in corresponding data and coordinate classes, `NDCubeBase`, `ExtraCoords`, and `GlobalCoords`. Unlike the ABCs, these classes are viable off-the-shelf tools for end users and serve as ideal parent classes for more specialized data objects. This has the advantage of centralizing the development and maintenance of functionalities common to all WCS-based, coordinate-aware ND data analysis in one open-source package. It aids discoverability and prevents wastage of resources on developing and maintaining the same functionalities in multiple packages. This can free developers to focus on specialized functionalities, enhancing their productivity.

Third, the `ndcube` package provides additional support for coordinate-aware manipulation and visualization of ND astronomical data. This is achieved through three high-level data classes, `NDCube`, `NDCubeSequenceBase/NDCubeSequence`, and `NDCollection`. `NDCube` combines `NDCubeBase` with a visualization suite and some additional features and so complies with the NDCube 2 API. The other classes are designed to handle multiple NDCube instances simultaneously. They do not comply with the NDCube 2 API but do follow the principle that all functionalities must be independent of the number of dimensions and combination of physical types.

##### 4.1. `NDCubeBase`, `NDCube`, and Visualization

`NDCubeBase` implements the NDCube 2 API. It is in an off-the-shelf class that can be imported and immediately employed by users. This is ideal for users and developers who do not want to reimplement the NDCube 2 API themselves.

<sup>25</sup> <https://docs.sunpy.org/projects/ndcube/en/stable/>



NDCubeBase is valid for any type of array that exposes shape and dtype attributes and can be indexed using the Python slicing API. It therefore supports a wide variety of array infrastructures including `numpy`, `dask`, etc. It is conceivable that a user with highly specific needs may prefer to implement their own NDCube2 API-compliant class from scratch. However, it is expected that the vast majority of users will be well-served by NDCubeBase or its slightly more advanced derivate, NDCube.

NDCube combines NDCubeBase with the default visualization suite and a limited number of additional features. NDCube is targeted at end users who want to access all NDCube functionalities with one easy import. By contrast, it is anticipated that developers are more likely to write their own visualization suites. Custom visualization suites can be easily registered with NDCube objects via a descriptor provided by the `ndcube` package. This enhances the versatility of NDCube objects.

The default visualization suite used by NDCube is called `MatplotlibPlotter`, which, as its name suggests, leverages the `matplotlib` package (Hunter 2007). It is not designed as a set of additional methods on NDCube but as a `Plotter` class that is registered via a descriptor provided by the `ndcube` package. Once a `Plotter` class is registered, it can be accessed via the `.plotter` attribute. This design has a number of advantages. First, it separates the APIs of the core data class and the visualization suite. Second, it makes the visualization suite pluggable, thus enabling users to register their own `Plotter`. Both these advantages are particularly important as visualization can be highly dependent on the type of data and the users' needs. Third, it enables `MatplotlibPlotter` to be registered with any NDCube2 API-compliant class, not just NDCube. Fourth, multiple visualization algorithms can be provided by the same instance as the `Plotter` can have any number of methods, e.g., `NDCube.plotter.viz_method1()`, `NDCube.plotter.viz_method2()`, etc. Fifth, it avoids the `ndcube` package having a required dependency on `matplotlib`. Instead, `MatplotlibPlotter` will only work if `matplotlib` is already installed. This enables developers to depend on `ndcube` without being forced to depend on such a heavyweight package as `matplotlib`. `MatplotlibPlotter` has a `.plot()` method that returns a plot or animation depending on the dimensionality of the cube and inputs of the user. By default, 2D image plots and animations are returned if the NDCube instance has more than 1D. However, line plots and animations can be returned instead if the user prefers. The plot axes are automatically labeled with the physical types and coordinates found in the coordinate transformations. As in the case of methods discussed in Section 3.2, users can select whether to use the primary WCS, `ExtraCoords`, or a combined WCS, by setting the `wcs=kwargs`.

Although `MatplotlibPlotter` does not fulfill all visualization needs, it is not anticipated that the `ndcube` package will provide more plotter classes for NDCube2 API-compliant objects. This is because visualization needs can be so specific to the type of data that not all cases can be anticipated or supported. Instead, `MatplotlibPlotter` acts as an example for developers on how to write their own `Plotter` class and as a default visualization suite for end users who just want to *quick-look* their data.

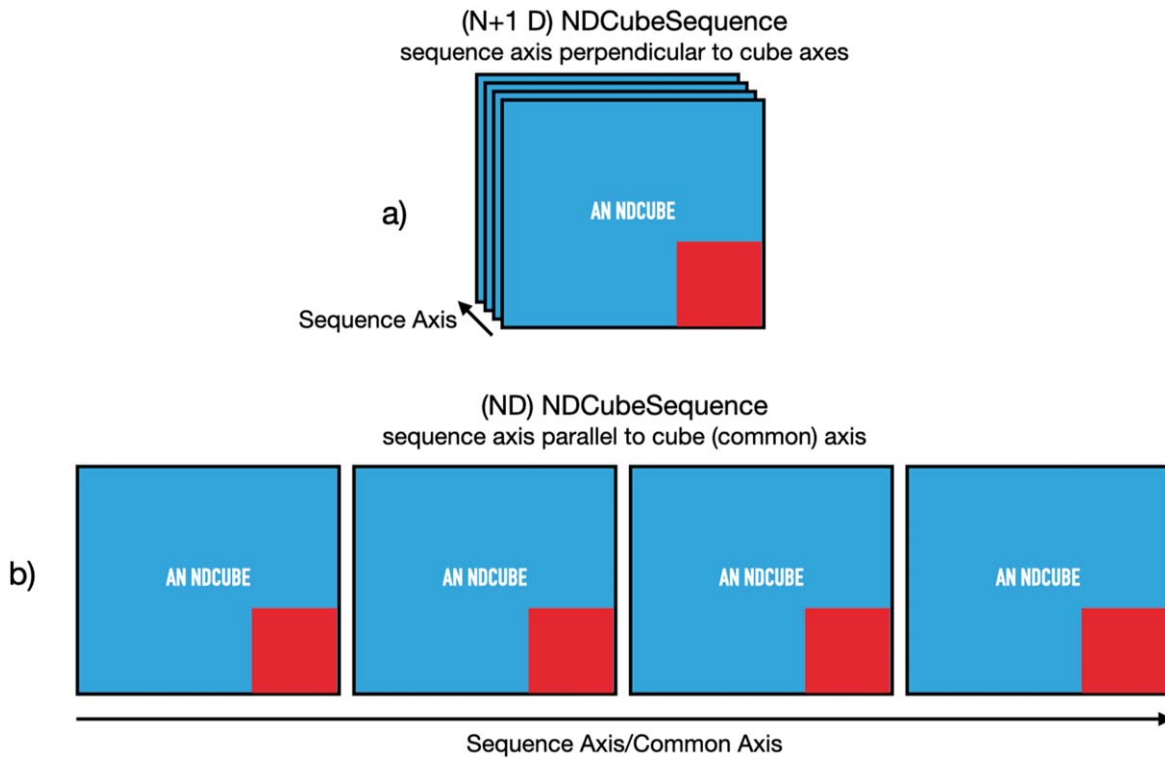
## 4.2. NDCubeSequence

NDCubeSequence combines multiple NDCube2 API-compliant instances as if they were one contiguous data set. The constituent cubes must be arranged in some order, have the same shape, and have the same physical types associated with each axis. It is also assumed that their data represent the same physical property. As an example, consider a sequence of images of the same region of the sky taken with the same instrument. Due to slight differences in the pointing jitter, each image has its own WCS. We could reproject these images to the same celestial WCS, add a time axis, and stack the images in a single NDCube object. However, if we want to avoid altering the data by reprojection or building a complex new WCS object, we could put each image in its own cube with its original WCS object and arrange them chronologically in an NDCubeSequence. The axis along which the constituent cubes are ordered is called the *sequence axis*. The NDCubeSequence API is designed so the sequence axis appears like an extra array axis to the user. So, in the example above, the user can interact with NDCubeSequence as if it were a 3D NDCube with almost the same API. For example, NDCubeSequence has its own slicing and cropping APIs that mimic those of NDCube. It enables the constituent cubes to be simultaneously cropped with a single command at the NDCubeSequence level.

In the above example, the cubes were ordered in time, which is a physical type not included among the world axes of their WCS transformations. In this case, the sequence axis acts as an additional *perpendicular* axis of the data set (Figure 3(a)). However, if the images were part of a horizontal mosaic, they would be ordered along the *x*-axis of the cubes. In this case, the sequence axis is said to be *parallel* to the *x*-axis (Figure 3(b)). The *x*-axis is referred to as the *common axis* because it is shared by the cubes. NDCubeSequence provides a set of methods and properties (prefixed with `cube_like_`) to allow users to interact with the sequence in this extended ND format. Say, as in Figure 3, we have four images each with a shape of  $(2, 3)$  where the axes are ordered  $(y, x)$ . Normally we would interact with this NDCubeSequence as if it were a single cube with a shape of  $(4, 2, 3)$ . However, if we set the common axis to the *x*-axis of the constituent cubes (`common_axis=1` in this example), NDCubeSequence's `cube_like_` API allows us to interact with it as if it were a cube of the shape  $(2, 12)$ . Because the regular and `cube_like_` APIs are separate, users can switch between them as required without any penalty. This is because the `cube_like_` API simply translates between the  $(N+1)$ -D and extended ND frameworks without altering the underlying data. This can be very useful if, for example, the images in the mosaic were taken sequentially in time, and users wanted to think of them as time-ordered in some cases and space-ordered in others.

Like NDCube, NDCubeSequence is actually composed of a base class, `NDCubeSequenceBase`, and a plotter class, `MatplotlibSequencePlotter`. `MatplotlibSequencePlotter` is much simpler than `MatplotlibPlotter`. It returns animations along the sequence axis and delegates the details of the animation to the `.plot` method of the constituent NDCube instances. If users or developers wish to dispense with visualization, they can use `NDCubeSequenceBase` instead of `NDCubeSequence`. Alternatively, they can use the sequence descriptor class provided by the `ndcube` package to register their own sequence plotter class with `NDCubeSequenceBase`.





**Figure 3.** Diagram of an NDCubeSequence with its sequence axis in the perpendicular (a) and parallel (b) configurations. The sequence axis is the axis along which the constituent cubes are ordered. In the perpendicular configuration, the sequence axis represents an additional axis to those of the cubes, e.g., a sequence 2D images ordered in time. In the parallel configuration, the cubes are ordered along one of their array axes, known as the *common axis*. An example would be a mosaic of images representing adjacent regions of the sky. NDCubeSequence provides APIs for interacting with the data in both configurations interchangeably.

### 4.3. NDCollection

NDCollection is a class for grouping NDCube2 API-compliant or NDCubeSequence(Base) instances. It differs from NDCubeSequence in that the objects contained are not ordered, are not assumed to represent measurements of the same physical property, and can have different dimensions. One application of NDCollection is linking observations with derived data products. Consider a 3D spectral image cube and a 2D Doppler velocity map derived from a spectral line in each pixel. These objects are clearly related. They share the same spatial axes. But they have different dimensionalities (3D and 2D), represent different physical properties (intensity and velocity), and do not have an order in their common coordinate space. They are therefore well-suited to being stored in an NDCollection. Each cube is referenced by its name, making NDCollection similar to a Python dictionary. However, it is more powerful than a dictionary in that it allows certain axes to be marked as *aligned*. While this is not required, it does facilitate simultaneous slicing of the collection’s components along those axes. In the above example, we could mark the spatial axes of cubes as *aligned* and then extract a spatial subregion by slicing the NDCollection rather than the two data objects separately (Figure 4). Thus, NDCollection enables easier and more reliable manipulation of ND astronomical data sets.

## 5. NDCube in Action

Figure 5 demonstrates the power of NDCube for handling real life high-dimensional data. The observations are of a solar active region taken by the Swedish Solar Telescope (SST;

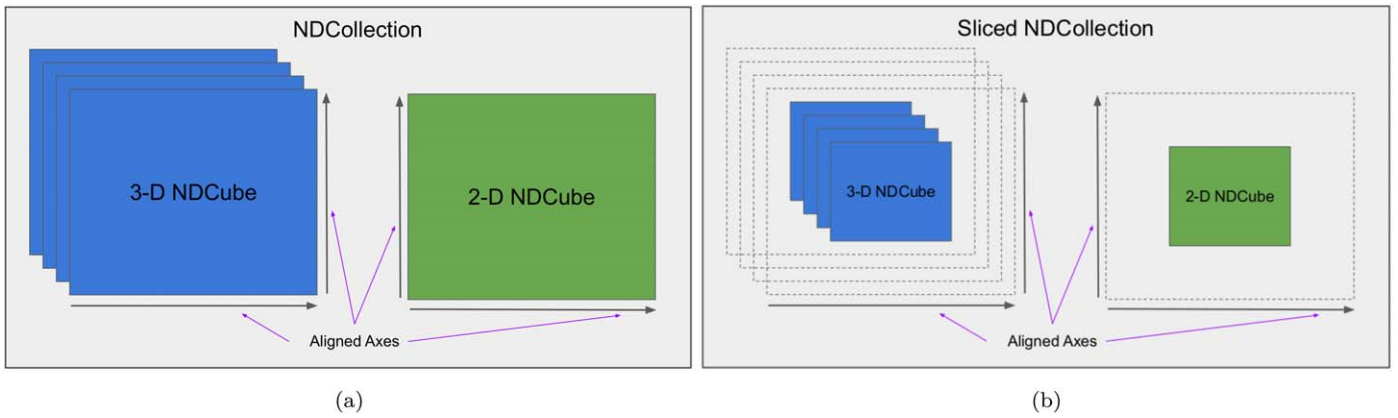
Scharmer et al. 2003; Scharmer 2017; Löfdahl et al. 2021). SST is a ground-based telescope that can take spectroscopic imaging observations at different polarization states as a function of time. Its observations are therefore often represented as 5D data cubes with axes of time, Stokes polarization parameter, wavelength, space, and space. Figure 5 shows observations at a single polarization state and so is only 4D. Other WCS-coordinate-aware data classes are designed to handle such data in limited ways, e.g., SunPy Map can only handle 2D spatial slices. However, NDCube can handle the entire data cube.

Figure 5(a) shows a single frame in an animation produced by the standard visualization suite. Although the visualization suite is not part of the NDCube2 API, it is available via the NDCube class and is useful here in demonstrating the utility of an NDCube2 API-compliant object. If the name of the NDCube instance is `sst_cube`, Figure 5(a) would be produced via the following:

```
1. >>> sst_cube.plot()
```

This defaults to a 2D animation with the last two axes plotted and the others represented by sliders below the plot. These can be used interactively to animate along unplotted axes. In this case, the last two axes are spatial, so an image animation is produced with sliders for wavelength and time. All the tick and axis labels are automatically generated by the visualization suite from the WCS object. This immediately orients the viewer and removes confusion as to which axis is which.

The animation frame shows a sunspot with its dark umbra and fibril-like penumbra. To explore the spectral or temporal



**Figure 4.** (a) An NDCollection containing 3D and 2D NDCube instances. The  $x$ - and  $y$ -axes are *aligned*, while the  $z$ -axis of the 3D NDCubeBase is not. (b) The result of applying the standard Python slicing API to the NDCollection in (a). Both cubes have been cropped along the  $x$ - and  $y$ -axes, while the 3D cube remains uncropped along the  $z$ -axis.

evolution of the sunspot at the location marked by the red dot, we extract that pixel via the standard slicing API:

```
1. >>> sst_spectrogram = sst_cube[:, :, y, x]
```

where  $y$  and  $x$  are the array indices of the pixel. We could also have achieved the same result with `NDCube.crop` if we wanted to input real world coordinates instead. The above operation extracts a 2D NDCube with axes of time and wavelength. Plotting this via

```
1. >>> sst_spectrogram.plot()
```

produces the spectrogram seen in Figure 5(b). As both dimensions are already represented in the plot, no animation sliders are produced. Each row of the spectrogram represents a spectrum at a certain time while each column represents a timeseries at a certain wavelength. To more easily inspect the temporal evolution at a certain wavelength (red vertical line, Figure 5(b)), we can again slice the cube and plot the result (Figure 5(c)):

```
1. >>> sst_timeseries = sst_spectrogram[:, w]
2. >>> sst_timeseries.plot()
```

where  $w$  is the array index corresponding to the chosen wavelength. In this case, the result is a 1D NDCube, so the resulting plot is a line. It shows that the intensity at that wavelength continually increases during the observations. Similarly, we can inspect the spectrum at a given time (red horizontal line Figure 5(b)) by slicing and plotting the other dimension of the spectrogram:

```
1. >>> sst_spectrum = sst_spectrogram[t]
2. >>> sst_spectrum.plot()
```

where  $t$  is the array index of the chosen time. The resulting spectrum in Figure 5(d) appears to show a broad absorption feature punctuated by smaller emission and/or absorption features. Note that we could also have produced a line animation of the spectrum as a function of time by using the `plot_axes` keyword argument when plotting `sst_spectrogram`:

```
1. >>> sst_spectrogram.plot(plot_axes=
    [None, 'x'])
```

This tells the visualization suite to assign the second cube axis to the  $x$ -axis on the plot but not to assign the first cube axis. As a result, a 1D line plot is produced with the time axis represented by an animation slider. This allows the user to dynamically view

how the spectrum changes with time, which may be more intuitive to some users than looking at a 2D spectrogram.

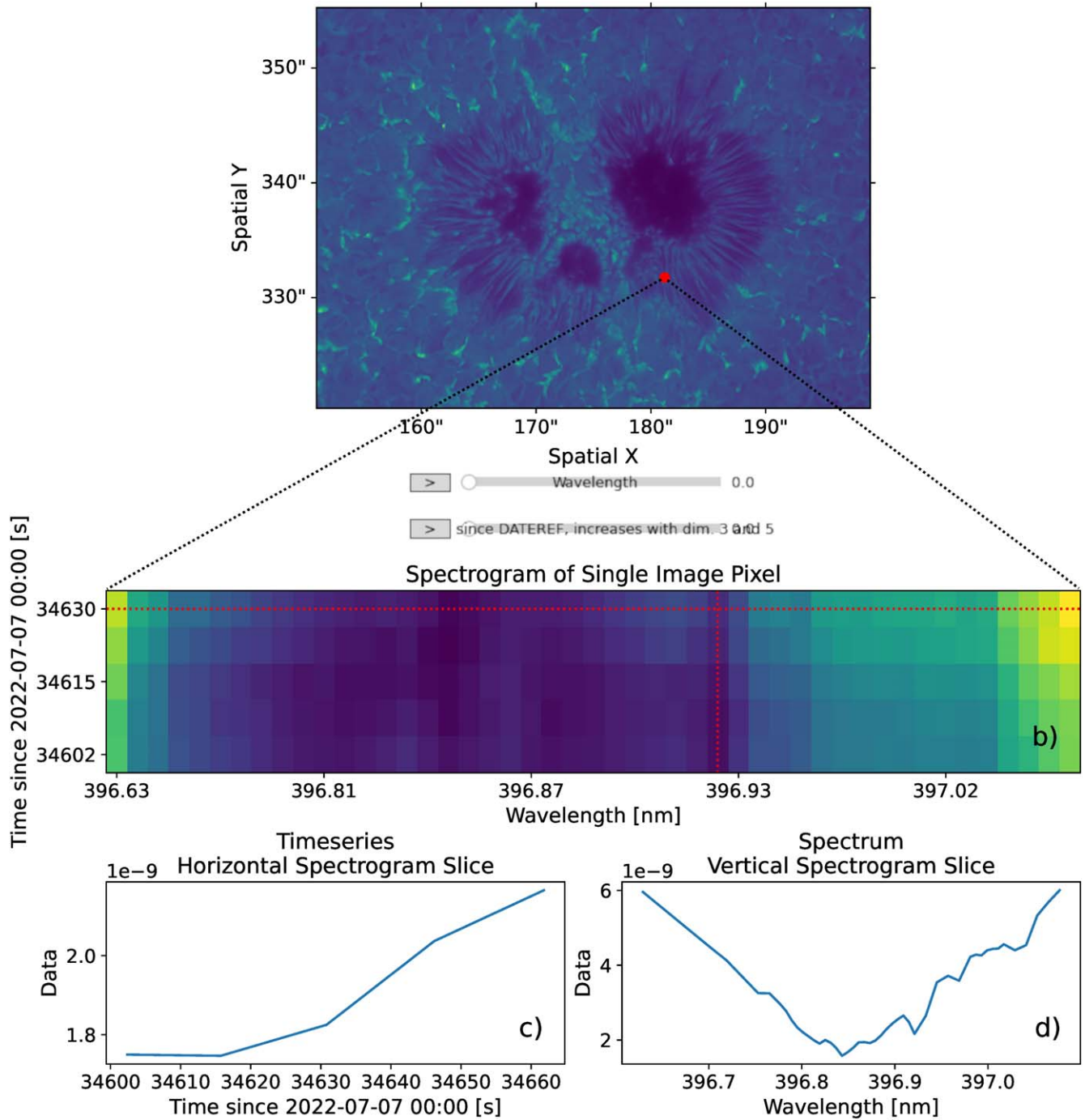
As already alluded to, the NDCube class provides several analysis tools in addition to the visualization suite, which are beyond the scope of the NDCube 2 API. For example, at the time of writing (`ndcube v2.1.1`), NDCube supports unit conversation similar to `astropy Quantity`'s `.to` method and basic arithmetic operations (negation, addition, subtraction, multiplication, and division) with scalars, arrays, and `Quantity` objects. `NDCube.rebin` can combine contiguous ND rectangles of array elements via various aggregation functions including sum and mean. This enables users to sacrifice resolution in 1D to boost signal-to-noise in another, a common practice in solar imaging spectroscopy. Additionally, the `NDCube.reproject_to` method enables reprojection to another set of valid WCS transformations. This can be used to convert observations from the different instruments onto the same pixel grid, or project the observations to the viewpoint of another observer. The latter is particularly helpful in the age of beyond-Earth-orbit satellite observatories such as STEREO and Solar Orbiter. Readers interested in learning more about these tools are encouraged to consult the `ndcube` documentation.<sup>26</sup>

## 6. Applications in the Community

The value of the NDCube 2 API and its implementation by the `ndcube` package is evidenced by the number of Python packages already depending on them. Some of these provide general analysis tools that support broad subsections of the astronomy community, while others support specific missions in the processing and analysis of their observations.

The Astropy coordinated package, `specutils`, provides tools for the manipulation and analysis of astronomical spectral observations (Astropy-Specutils Development Team 2019). Its fundamental data container, `Spectrum1D`, inherits from NDCube. The 1D in its name refers to the fact that, by design, it supports data with a single spectral dimension. However, it can support any number of additional nonspectral dimensions and inherits functionalities from NDCube such as slicing. Through `specutils`, NDCube also supports the data cube handling in `Jdaviz`, a data analysis and visualization package for JWST (JDADF Developers et al. 2023).

<sup>26</sup> <https://docs.sunpy.org/projects/ndcube/en/stable/>



**Figure 5.** Visualizations of different slices of an NDCube holding 4D time–wavelength–space–space observations of a sunspot taken by the Swedish Solar Telescope. Each panel was produced with two simple commands, one to slice the NDCube and another to plot the result. (a) A frame from an animation of the 4D cube with the spatial axes plotted and time and wavelength represented by animation sliders. (b) 2D spectrogram produced by extracting the pixel marked by the red dot in panel (a). (c) The timeseries corresponding to the vertical red line in panel (b). (d) The spectrum corresponding to the horizontal red line in panel (b).

SunPy’s `sunraster` package is designed to manipulate, analyze, and visualize observations from rastering slit spectrographs (Ryan et al. 2023b). These instruments disperse light through a slit and build up spectral image cubes by scanning the slit over a region of interest and taking sequential spectra. This is known as rastering. The resulting data cubes have dimensions of time–wavelength–space–space/time. One spatial dimension is coupled with time due to the rastering process while a related time dimension represents repeated raster scans. The `sunraster` data containers inherit NDCube and NDCubeSequence. The

NDCubeSequence.cube\_like\_ APIs (Section 4.2) can be used to seamlessly switch between different dimensional representations, namely, time–wavelength–space and time–wavelength–space–space. In addition, the `ExtraCoords` API enables the time coordinates to be kept separate from, or combined with, the primary WCS transformations as desired. While most of its tools are instrument-agnostic, `sunraster` provides specific support for Solar Orbiter/Spectral Imaging for the Coronal Environment (SPICE; SPICE Consortium et al. 2020). SPICE, in combination with other instruments on board Solar Orbiter, aims



to improve our understanding of the complex connection between the Sun and the inner heliosphere by observing various EUV spectral lines. The SPICE instrument team uses `sunraster` to read, manipulate, and visualize their data.

Other solar UV–EUV slit spectrographs currently in operation include the Hinode/EUV Imaging Spectrograph (EIS; Culhane et al. 2007) and the Interface Region Imaging Spectrograph (IRIS; De Pontieu et al. 2014). These instruments are improving our understanding of the complex mechanisms that drive and sustain the solar chromosphere and low corona by inferring bulk flows, thermodynamics, and plasma composition from UV and EUV imaging and spectroscopy. The IRIS Python user tools (`irispy-lmsal`: [irispy-lmsal Development Team 2023](https://irispy-lmsal.readthedocs.io/en/stable/)) and the EIS Python Analysis Code (EISPAC: [Weberg et al. 2023](https://eispy.readthedocs.io/en/latest/))<sup>27</sup> depend heavily on `sunraster` and `ndcube`. In addition to the plotting and coordinate capabilities, the flexible `.meta` dictionary and subclass extensibility of `ndcube` objects enable EISPAC to conveniently and efficiently store EIS-specific ancillary data and functions. This improves the general accessibility and analysis capability for the current database of over 430,000 EIS observations.

The NDCube 2 API is not only suited to spectral observations. The Polarimeter to UNify the Corona and Heliosphere (PUNCH; Deforest et al. 2022), scheduled for launch in 2025, is a constellation mission that will reveal how the solar corona influences the environment throughout the heliosphere via polarimetric observations. The core data container of its processing pipeline is `NDCube`, bundling a data array with uncertainties and a WCS object. `NDCube` provides a base framework for a PUNCH data object that supports the complexity of multiple spacecraft observations, while interfacing with `SunPy` and `Astropy` libraries, such as `reproject`.

The Daniel K. Inouye Solar Telescope (DKIST; Rimmele et al. 2020) is one of the solar community’s flagship ground-based observatories. Its user tools depend on the NDCube 2 API and facilitate analysis of its spectral, polarimetric, and imaging observations. Due to the vast amounts of data produced by DKIST, one of the attractions of the NDCube 2 API to the DKIST team is its implicit support for `dask` arrays. Unlike more traditional array packages like `numpy`, `dask` supports lazy data loading and operations, which enable analysis workflows to be developed before data is read into memory. Once the time comes to execute, `dask` supports parallelization of the workflow in a user-friendly way. The combination of `dask` and the NDCube 2 API enables users to manipulate this vast amount of data in almost the same way as they would small-data observations in their laptops’ RAM.

The NDCube 2 API and `ndcube` package are expected to support more packages in the future. For example, the `sunpy` (Mumford et al. 2020) and `spectral_cube` (Ginsburg et al. 2019)<sup>28</sup> packages are currently planning to refactor their flagship data containers to depend on `NDCube`, namely, `Map` for 2D solar images and `SpectralCube` for space–space–spectral cubes with an optional additional Stokes dimension, respectively. This will make their APIs compatible with other NDCube 2 API-compliant classes serving different types of data. It will also enable support for additional nonspatial dimensions similar to how `specutil`’s `Spectrum1D` supports additional nonspectral dimensions. This opens the

possibility of merging the roles of `Map` with `SunPy`’s `MapSequence` class for chronologically ordered image stacks. Alternatively, `MapSequence` could inherit from `NDCubeSequence`, which provides a more powerful suite of analysis methods than does `MapSequence` currently.

The continued proliferation of the NDCube 2 API will make it simpler for scientists to jointly analyze different types of coordinate-aware data as common tasks will be performed in the same way. This will lower the barriers to multiinstrument, interdisciplinary studies and boost scientific output of the broader community and agency-funded missions.

## 7. Future of the `ndcube` Package

The release of `ndcube` v2.1.1 marks the attainment of many goals on the `ndcube` roadmap. However, there is plenty of scope for further development. Arithmetic support could be expanded to include `NDCubeSequence`, raising `NDCube` instances to scalar powers, and operations between `NDCube` and coordinate-less `astropy` `NDData` instances. Support between `NDCube` and coordinate-aware classes is considered a longer term goal due to the ambiguity of combining data with potentially different coordinate transformations.

A new `NDCubeSequence` method could be developed to reproject its constituent cubes to a common WCS, then stack them into an `NDCube` instance. This would make it much easier for users to combine cubes with slightly different WCS transformations and access the benefits of storing and manipulating data in a single `NDCube`.

Conversion methods between `ndcube` and `xarray` classes would enhance interoperability between scientific fields and use-cases that do and do not require WCS coordinate transformations. The `ndcube` and `xarray` teams have already held discussions on this topic.

Support for functional uncertainties and masks could greatly help memory efficiency for large data sets. A more speculative concept is the development of a functional or array-based mask collection class, which would enable users to simultaneously store multiple masks. These could be used to represent different phenomena, e.g., cosmic ray hits, detector artifacts, celestial features of scientific interest, etc., and could be applied to the data individually or in combination as the user desired.

Development of methods to write `ndcube` data classes to file would greatly help to preserve the output of computationally demanding workflows and make sharing partially completed analysis easier, thereby facilitating collaborations.

Better handling of axis-specific metadata is an important future development for `ndcube`. Axis-specific metadata—e.g., variable exposure time of images in an image cube—do not necessarily vary monotonically along the array axis. They are therefore not coordinates and cannot be stored in a WCS or `ExtraCoords` object. However, storing them in the metadata attribute can lead them to becoming inconsistent with the cube’s dimensionality since the metadata object is currently not sliced by the data class’ slicing infrastructure. A sliceable metadata object would greatly aid the storage and manipulation of such information.

These ideas and more are currently being considered for the next iteration of the `ndcube` roadmap. Readers who feel strongly about any of these features, or others not listed here, are invited to contact the `ndcube` community via the channels outlined in the `ndcube` documentation<sup>29</sup> or our GitHub Issue

<sup>27</sup> <https://irispy-lmsal.readthedocs.io/en/stable/>

<sup>28</sup> <https://spectral-cube.readthedocs.io/en/latest/>

<sup>29</sup> <https://docs.sunpy.org/projects/ndcube/en/stable/index.html>

Tracker.<sup>30</sup> We are always interested in hearing more about the needs of users, and we particularly welcome those interested in contributing to the next version of `ndcube`.

## 8. Summary

The NDCube 2 API enables users to store, manipulate, and inspect astronomical coordinate-aware data in a standardized way, analogous to how coordinate-agnostic data is treated in arrays. It depends heavily on the AstroPy WCS API (Section 2), which provides a standardized way to interact with WCS transformations, regardless of their underlying implementation. Together, these APIs lead to more intuitive and reliable analysis workflows. This enables scientists to move closer to the speed of thought and boost their scientific output.

The `ndcube` package (Section 4) formalizes the NDCube 2 API (Section 3) as defined by Mumford & Ryan (2020) via its ABCs NDCubeABC, ExtraCoordsABC, and GlobalCoordsABC. This API is implemented in a directly usable form by the NDCubeBase, ExtraCoords, and GlobalCoords classes. The NDCube class inherits from NDCubeBase and provides some additional API features including a pluggable visualization suite. Additional classes, NDCubeSequence and NDCollection, provide ways of linking and jointly manipulating multiple objects that comply with the NDCube 2 API. While these classes are powerful end-user tools in their own right (Section 5), they are also ideal as parent classes for building more targeted tools. This is evident from the number of Python packages that now depend on the NDCube 2 API and the classes in the `ndcube` package (Section 6). These packages serve broad sections of the astronomy community as well as specific flagship missions such as JWST and Solar Orbiter. For highly specific use-cases, the same benefits can be provided to the end user by developing classes with their own underlying implementation but that still adhere to the NDCube 2 API. This is currently the case for the DKIST user tools.

With the release of version (2.1.1), the `ndcube` package has reached a level of stability upon which users can depend. That said, the `ndcube` package continues to be enhanced (Section 7). Readers who have ideas about how the `ndcube` package can better meet users' needs and/or those who would like to contribute to the next versions of the package are encouraged to contact the `ndcube` community via the channels listed in the documentation (see footnotes (5) and (6)).

The continuing proliferation of NDCube 2 API and the `ndcube` package is due to their power and versatility. They liberate users and developers from executing common, well-defined but tedious and error-prone tasks, in a way that is valid for any type of ND data described by an AstroPy-WCS-API-compliant set of coordinate transformations. This promotes scientific output and more efficient software development. Moreover, they promote API convergence across packages, which reduces technical barriers and promotes multiinstrument studies and cross-community collaborations.

## Acknowledgments

We thank NASA's Heliophysics Data Environment Enhancement program, the Daniel K. Inouye Solar Telescope,

and Solar Orbiter/SPICE (grant No. 80NSSC19K1000) for financial support. We also thank the SunPy, AstroPy, and Python in Heliophysics communities for their support and engagement. The SST observations are thanks to Tiago Pereira, Reetika Joshi, Kilian Krikova, Ana Belen Griñón Marin, and Luc Rouppe van der Voort. The Swedish 1 m Solar Telescope is operated on the island of La Palma by the Institute for Solar Physics of Stockholm University in the Spanish Observatorio del Roque de los Muchachos of the Instituto de Astrofísica de Canarias. The Institute for Solar Physics is supported by a grant for research infrastructures of national importance from the Swedish Research Council (registration No. 2017-00625).

## Appendix A Terminology

### A.1. World, Pixel, and Array Axes

Throughout this paper, we use the terms *world*, *pixel*, and *array* to describe coordinates, axes, and axis ordering. A set of WCS transformations can describe any number of physical types, e.g., time, latitude, longitude, location along the electromagnetic spectrum, etc. These are referred to as world axes, and the order in which they are stored in the WCS object is referred to as world order (or world axis order). These physical types are mapped through the WCS to one or more *pixel* axes. Although this name is taken from the pixels of a camera detector, the term *pixel axes* refers more generally to the axes of the array in which the data is stored, irrespective of the type of instrument that generated them. Hence, an element in a timeseries or spectrum array can be referred to as a pixel, even if no real pixels were used in making the measurement.

Although, in the simplest case, one world axis would uniquely map to one pixel axis, it is possible that world axes be associated with multiple pixel axes and vice versa. This is especially common when dealing with projections of the sky onto 2D image planes. For example, consider an image of the Earth taken from space. There is not one pixel axis for latitude and another for longitude. Instead, as we move vertically or horizontally across the image, both latitude and longitude will vary, no matter how we orient the Earth in the image. In this case, latitude and longitude are referred to as *dependent world axes*, or simply *dependent*.

Due to historical precedent, WCS pixel axes are column-major. However, `numpy` arrays are row-major. This leads to the confusing scenario where the first WCS pixel axis corresponds to the last `numpy` array axis, the second pixel axis corresponds to the second to last `numpy` array axis, and so on. We therefore use the term *array axes* to refer to the pixel axes that have been reversed to reflect the axis order of the `numpy` data array.



















### A.2. Axis Correlation Matrix

In a WCS object, the mapping between pixel and world axes is described by the axis correlation matrix, a 2D boolean array whose columns represent pixel axes and rows represent world axes. The `True/False` value at a specific row-column index states whether that world axis maps to that pixel axis. Thus, the axis correlation matrix can represent simple mappings where each world axis uniquely maps to one pixel axis (e.g., a diagonal matrix), or a more complex case where world axes map to multiple pixels axes and/or vice versa (multiple `true` values in each row and/or column). While many users will not

<sup>30</sup> <https://github.com/sunpy/ndcube/issues>

have to deal directly with the axis correlation matrix, it is crucial to the underlying WCS infrastructure and its advanced usage.

### ORCID iDs

Daniel F. Ryan  <https://orcid.org/0000-0001-8661-3825>  
 Stuart Mumford  <https://orcid.org/0000-0003-4217-4642>  
 Will T. Barnes  <https://orcid.org/0000-0001-9642-6089>  
 Adwait Bhope  <https://orcid.org/0000-0002-7133-8776>  
 Éric Buchlin  <https://orcid.org/0000-0003-4290-1897>  
 Nabil Freij  <https://orcid.org/0000-0002-6253-082X>  
 Adam Ginsburg  <https://orcid.org/0000-0001-6431-9633>  
 Laura A. Hayes  <https://orcid.org/0000-0002-6835-2390>  
 Derek Homeier  <https://orcid.org/0000-0002-8546-9128>  
 J. Marcus Hughes  <https://orcid.org/0000-0003-3410-7650>  
 Chris Lowder  <https://orcid.org/0000-0001-8318-8229>  
 Richard O'Steen  <https://orcid.org/0000-0002-2432-8946>  
 Thomas Robitaille  <https://orcid.org/0000-0002-8642-1329>  
 David Stansby  <https://orcid.org/0000-0002-1365-1908>  
 Albert Y. Shih  <https://orcid.org/0000-0001-6874-2594>  
 Erik Tollerud  <https://orcid.org/0000-0002-9599-310X>  
 Micah J. Weberg  <https://orcid.org/0000-0002-4433-4841>  
 Matthew J. West  <https://orcid.org/0000-0002-0631-2393>

### References

- Astropy Development Team (2023) The astropy Code-base, GitHub, <https://github.com/astropy/astropy>
- Astropy-Specutils Development Team (2019) Specutils: Spectroscopic analysis and reduction, Astrophysics Source Code Library, ascl:1902.012
- Astropy Collaboration, Price-Whelan, A. M., Sipőcz, B. M., et al. 2018, *AJ*, 156, 123
- Astropy Collaboration, Robitaille, T. P., Tollerud, E., et al. 2013, *A&A*, 558, A33
- Calabretta, M. R. (2011) Wcslib and Pgsbox, Astrophysics Source Code Library, ascl:1108.003
- Calabretta, M. R., & Greisen, E. W. 2002, *A&A*, 395, 1077
- Culhane, J. L., Harra, L. K., James, A. M., et al. 2007, *SoPh*, 243, 19
- De Pontieu, B., Title, A. M., Lemen, J. R., et al. 2014, *SoPh*, 289, 2733
- Deforest, C., Killough, R., Gibson, S., et al. 2022, in 2022 IEEE Aerospace Conf. (Piscataway, NJ: IEEE), 1
- Dencheva, N., & Greenfield, P. 2019, in ASP Conf. Ser. 523, Astronomical Data Analysis Software and Systems XXVII, ed. P. J. Teuben et al. (San Francisco, CA: ASP), 535
- Dencheva, N., Mumford, S., Cara, M., et al. 2023, spacetelescope/gwcs: GWCS v 0.18.3 v0.18.3, Zenodo, doi:10.5281/zenodo.7478201
- Ginsburg, A., Koch, K., Robitaille, T., et al. 2019, radio-astro-tools/spectralcube: Release v0.4.5 v0.4.5, Zenodo, doi:10.5281/zenodo.3558614
- Greisen, E. W., & Calabretta, M. R. 2002, *A&A*, 395, 1061
- Greisen, E. W., Calabretta, M. R., Valdes, F. G., & Allen, S. L. 2006, *A&A*, 446, 747
- Hanisch, R. J., Farris, A., Greisen, E. W., et al. 2001, *A&A*, 376, 359
- Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, *Natur*, 585, 357
- Hoyer, S., & Hamman, J. 2017, *JORS*, 5, 10
- Hunter, J. D. 2007, *CSE*, 9, 90
- irisy-lmsal Development Team 2023, The irisy-lmsal Code-base, GitHub, <https://github.com/LM-SAL/irisy-lmsal>
- Ivezić, Ž., Kahn, S. M., Tyson, J. A., et al. 2019, *ApJ*, 873, 111
- JDADF Developers, Averbukh, J., Bradley, L., et al. 2023, Jdaviz, v3.3.0, Zenodo, 10.5281/zenodo.7625637
- Löfdahl, M. G., Hillberg, T., de la Cruz Rodríguez, J., et al. 2021, *A&A*, 653, A68
- Mumford, S., Freij, N., Christe, S., et al. 2020, *JOSS*, 5, 1832
- Mumford, S., & Ryan, D. F. 2020, SunPy Proposal for Enhancement 12: NDCube 2 (SEP 0012), v1, Zenodo, doi:10.5281/zenodo.7020103
- ndcube Development Team 2023, The ndcube Code-base, GitHub, <https://github.com/sunpy/ndcube>
- Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. 2017, in Proc. Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conf. Neural Information Processing Systems (NIPS) [http://learningsys.org/nips17/assets/papers/paper\\_16.pdf](http://learningsys.org/nips17/assets/papers/paper_16.pdf)
- Pence, W. D., Chiappetti, L., Page, C. G., Shaw, R. A., & Stobie, E. 2010, *A&A*, 524, A42
- Rimmele, T. R., Warner, M., Keil, S. L., et al. 2020, *SoPh*, 295, 172
- Robitaille, T., Tollerud, E., Mumford, S., & Ginsburg, A. 2018, Astropy Proposal for Enhancement 14: A shared Python interface for World Coordinate Systems (APE 14) v1, Zenodo, doi:10.5281/zenodo.1188875
- Rots, A. H., Bunclark, P. S., Calabretta, M. R., et al. 2015, *A&A*, 574, A36
- Ryan, D. F., Freij, N., Mumford, S., et al. 2023b, *JOSS*, submitted
- Ryan, D. F., Mumford, S., Sharma, Y., et al. 2023a, *JOSS*, 8, 5296
- Scharmer, G. 2017, SOLARNET IV: The Physics of the Sun from the Interior to the Outer Atmosphere, 85
- Scharmer, G. B., Bjelksjo, K., & Korhonen, T. K. 2003, *Proc. SPIE*, 4853, 341
- SPIEC Consortium, Anderson, M., Appourchaux, T., et al. 2020, *A&A*, 642, A14
- Weberg, M., Warren, H., Crump, N., & Barnes, W. 2023, *JOSS*, 8, 4914
- Wells, D. C., Greisen, E. W., & Harten, R. H. 1981, *A&AS*, 44, 363