191215-02-proposal

December 31, 2019

1 Proposal

1.1 Named Array Backend

Summary based on Exploring Julia xarray equivalents notebook.

There are currently around five options in Julia for creating n-dimensional arrays with named axis:

- AxisArrays
- NamedArrays
- NamedDims.jl
- DimensionalData.jl
- ITensors.jl

Out of those, both DimensionalData and ITensors explicitly stated that they are preview releases/very early in development and that the interfaces will continue to change. This is an inevitable problem in Julia as the language itself is very young and the ecosystem has yet to settle on standard packages in the same way that Python has.

The remaining options were AxisArrays, NamedArrays, and NamedDims. The main benefits Julia has over Python is performance and the focus on multiple dispatch, these both require type-stable data that the compiler can easily deal with and interpret, NamedArrays does not allow for this when doing axis lookups so (even though it is in the top for popularity) I decided to rule it out.

NamedDims does what it says on the tin and lets you name the dimensions - without associated coordinate values, so it was ruled out.

Which leaves AxisArrays. However, as mentioned in accompanying notebook, there are extensive discussion about implementing large changes to AxisArrays in the future due to some limitations caused by the current architecture of the package. These changes should end up being mostly internal, meaning that packages depending on AxisArrays should not have to change much (if at all) to be compatible with future versions. The discussions are spread across these issues on GitHub: AxisArrays Roadmap, AxisArrays Issue: Use value indexing by default, AxisArraysFuture Plan.

Given those discussions, I had another look at DimensionalData (which explicitly mentions that it is under active development and unstable) and it is also a very nice option, as it is effectively a much newer 'cleaner' version of AxisArrays. Its implementation allows for much easier extensibility and abstraction, the syntax is more user-friendly and slightly less verbose, and finally it appears to solve the problems AxisArrays has encountered.

In the end stability will be a problem with any choice in an ecosystem which has not settled down on standard packages. As an indication of how much movement is planned, here is a comment from a discussion about the future of many of these packages:

> I just started looking through the NamedDims repo, so I apologize if this is already documented but I missed it. Is NamedDims ultimately intended to be integrated into AxisArrays or is it suppose to be a dependency or something else entirely?
>
> NamedDims.jl is intended to: A) Be used on its own, B) Be integrated into a future package along with IndexedDims.jl (name pending, Indexes.jl?), and that future package will replace AxisArrays.jl. (Like how StaticArrays.jl, replaced FixedSizedArrays.jl)

My honest view is that this space will change massively over the next few years, so no current choice will survive for long. There are a lot of very new, very early-development packages like AbstractIndecies or IndexedDims, AcceleratedArrays, DimensionalArrayTraits and more, which have the potential to completely change the ecosystem in the future depending on their success. These packages range from being functional, but very early in development and highly unstable, to conceptual with only a rough API defined.

A caveat to this is that even if things do change that should not be too big of a problem. Unlike Python, Julia has an excellent built-in package manager which handles version dependencies and concretisation beautifully, meaning large deprecations should not be a problem as you will always know what versions your package depends on and you can install all of these in a separate environment very easily.

Given all I have mentioned about how unstable the ecosystem currently is, it would be nice to have a system with a very flexible array backend. This has been done for a number of 'metapackages' like Plots.jl, where the package itself is an interface over multiple libraries. An ideal solution would be to be able to be able to swap the array backend with a single command similar to how it is done in Plots.jl, this way adapting to changes in the future would only require adding in a new set of interface methods which dispatch to different array constructors and methods.

The other bonus is that all of these packages are relatively simple wrappers around the base Julia implementation of arrays, meaning that they are all on the order of 1-2 thousand lines of code, whereas xarray is on the order of tens of thousands of lines (core being ~28k). This isn't the best measure of complexity, but such a huge difference does indicate a big difference in the amount of effort and knowledge required to maintain these packages, which is a good sign for the long-term health of the ecosystem (at least once it has had a chance to settle down).

In summary, there's no easy way to pick the best option or to say what will be used in a year or two. But, at least for now, AxisArrays has the most good points:

- It has a larg set of contributors, and more use in the community
- The interface is quite similar `xarrays`
- It allows for "type-stable selection of dimensions and compile-time axis lookup", which lets the compiler to keep the code performant
- Already has integration with other packages, e.g. `SimpleTraits.jl`
- Implementation with a metadata layer exists in `ImageMetadata.jl` - required for xarray-like attributes

1.2 Julia Mockup

Loading a data in cfgrib currently produces an xarray Datasets as such:

```
<xarray.Dataset>
Dimensions:         (isobaricInhPa: 2, latitude: 61, longitude: 120, number: 10, time: 4)
Coordinates:
  * number          (number) int64 0 1 2 3 4 5 6 7 8 9
  * time            (time) datetime64[ns] 2017-01-01 ... 2017-01-02T12:00:00
    step            timedelta64[ns] ...
  * isobaricInhPa   (isobaricInhPa) int64 850 500
  * latitude        (latitude) float64 90.0 87.0 84.0 81.0 ... -84.0 -87.0 -90.0
  * longitude       (longitude) float64 0.0 3.0 6.0 9.0 ... 351.0 354.0 357.0
    valid_time      (time) datetime64[ns] ...
Data variables:
    z               (number, time, isobaricInhPa, latitude, longitude) float32 ...
    t               (number, time, isobaricInhPa, latitude, longitude) float32 ...
Attributes:
    GRIB_edition:            1
    GRIB_centre:             ecmf
    GRIB_centreDescription:  European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             European Centre for Medium-Range Weather Forecasts
    history:                 2019-12-15T15:28:26 GRIB to CDM+CF via cfgrib-0....
```

The Julia implementation using AxisArrays would look like:

```
5-dimensional AxisArray{Float32,5,...} with axes:
    :number, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    :time, DateTime[2017-01-01T00:00:00, 2017-01-01T12:00:00, 2017-01-02T00:00:00, 2017-01-02T1
    :isobaricInhPa, [850, 500]
    :latitude, [90.0, 87.0, 84.0, 81.0, 78.0, 75.0, 72.0, 69.0, 66.0, 63.0  …  -63.0, -66.0, -6
    :longitude, [0.0, 3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0, 27.0  …  330.0, 333.0, 336.0

And data
     z, a 10×4×2×61×120 Array{Float32,5}
     t, a 10×4×2×61×120 Array{Float32,5}

With properties
    GRIB_edition: 1
    GRIB_centre: ecmf
    GRIB_centreDescription: European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre: 0
    Conventions: CF-1.7
    institution: European Centre for Medium-Range Weather Forecasts
    history: 2019-12-15T15:28:26 GRIB to CDM+CF via cfgrib-0....
```

Although the `step` and `valid_time` coordinates have been left out for now as it is not clear where

the best place to store them is in `AxisArrays`.

Similar to xarray, you would then access the `DataArray` equivalent by:

```
> ds.z
 5-dimensional AxisArray{Float32,5,...} with axes:
     :number, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
     :time, DateTime[2017-01-01T00:00:00, 2017-01-01T12:00:00, 2017-01-02T00:00:00, 2017-01-02T
     :isobaricInhPa, [850, 500]
     :latitude, [90.0, 87.0, 84.0, 81.0, 78.0, 75.0, 72.0, 69.0, 66.0, 63.0  …  -63.0, -66.0, -
     :longitude, [0.0, 3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0, 27.0  …  330.0, 333.0, 336

 And data, a 10×4×2×61×120 Array{Float32,5}:
 [:, :, 1, 1, 1] =
  252.663  251.854  251.142  252.044
  252.277  251.73   250.983  252.548
  252.449  251.733  250.829  252.358
  252.283  252.258  250.811  252.494
  252.049  251.622  250.824  251.921
  252.376  252.039  251.123  252.284
  252.131  251.842  251.281  252.17
  252.173  251.64   251.116  252.252
  251.714  251.768  251.422  252.368
  251.881  251.83   250.935  252.599
  ...

 With properties
     GRIB_typeOfLevel: isobaricInhPa
     long_name: Temperature
     GRIB_dataType: an
     GRIB_totalNumber: 10
     GRIB_jScansPositively: 0
     ...
```

From here data access is as specified in the AxisArrays documentation

I have not given much thought to the the coordinate transformation functions in cfgrib can apply via `cf2cdm`, but as far as I can see adding this in should not be a problem.

1.3 Calling ecCodes

Julia has built-in support for calling C libraries directly, so writing an interface to ecCodes is dooable. Additionally Julia has a very convenient `BinDeps`/`CondaBinDeps` package which can handle binary dependencies.

However, as mentioned in this notebook there is already a GRIB package in Julia which looks like it may be a nice starting point. It also provides a convenient `Index` type which can be used to filter the messages so that you only load data you are interested in.

This `Index` type looks to be very similar to the heterogeneous filtering cfgrib can perform with the `backend_kwargs={'filter_by_keys': {'typeOfLevel': 'surface'}}` arguments.

If it is actually feasible to build on top of this GRIB package is not clear to me yet, but it is an option I'd explore when working on the package.

1.4 Summary

1.4.1 Array Backend

In summary, AxisArrays was chosen as the Julia xarray equivalent, although how the ecosystem will evolve in the future is not clear so the Julia implementation will be developed with that in mind and will make it as easy as possible to swap out the array backend.

A few features will need to be added on top of AxisArrays to achieve the same functionality as xarray:

- `DataSet` (multiple `DataArrays`) support:
  - Currently AxisArrays only store one set of data - equivalent to a `DataArray` in xarray
  - xarray has the notion of a `DataSet`, which is a group of multiple `DataArrays` with common coordinates
  - This does not exist in AxisArrays but is quite easy to add in with a wrapper
- Metadata (`attrs`) support
  - Can be achieved with a simple wrapper function, or could be done via integration with `ImageMetadata.jl`
- Non-dimension coordinate support
  - xarray can store coordinates used for auxiliary labeling, e.g. the `step` and `valid_time` coordinates
  - This does not exist, but can be added in with a wrapper

All of these are features which should exist (and already might as I have not used AxisArrays that much, and the documentation is not extremely thorough) already, so my plan would be to first develop them working on this project, and then to request to merge them into AxisArrays. This would help build the community more, and also help with long-term code stability.

1.4.2 cfgrib Features

Calling the ecCodes is easy to do through Julia as C can be called directly, however I would prefer to integrate with and support existing packages where possible, so I would start off by building on top of the existing GRIB package in Julia.

This package is (as per their readme) "an interface to the ECMWF ecCodes library", where "a GribFile functions similarly to a Julia IOStream, except that instead of working as a stream of bytes, GribFile works as a stream of messages."

GRIB.jl already supports the concept of an `Index` type, which can be used to filter the messages so that you only receive the data you are interested in - this is similar to the heterogeneous filtering cfgrib can perform with the `backend_kwargs={'filter_by_keys': {'typeOfLevel': 'surface'}}` arguments.

Another feature of cfgrib which needs supported is the notion of coordinate transformations, which I have not looked at much, but from what I understood of the cfgrib documentation there won't be any problems implementing similar functionality in Julia.

1.4.3 Next Steps

My proposed plan starting in January is:

1. Get some examples of moderately complex real-life uses of cfgrib, along with the required data, which use all of the features of cfgrib. I would use these examples to create some tests, which would be the starting point of the development of the Julia package.
2. Work on the ecCodes calls, either via `GRIB.jl` or with a new implementation from scratch, get this to a functional level where the files can be read in as standard Julia arrays and dictionaries
3. Work on the wrapper for AxisArrays, adding in support for:
    1. `DataSet`s - multiple collections of `AxisArray`s
    2. Metadata - dictionary attached to a ArraySet/AxisArray, like `attrs` for xarray
    3. Non-dimensional coordinates
4. Test development done in parallel to each of the above stages
    1. Finalise unit tests
    2. Finalise integration tests and example tests
5. Potential tests which use `PyCall` to compare the results between Julia and Python
6. If possible, merge some code back into AxisArrays.jl and GRIB.jl
7. Nice to have features/integrations:
    1. `SimpleTraits`
    2. `FileIO`
    3. `JuliaDB`-like distributed multiprocessing