

## API for handling limit values at axis

A good API for handling axis values should

1. Compute the limit of a quantity only when the grid has a node at  $\rho = 0$ .
2. Compute limit dependencies only when the grid has a node at  $\rho = 0$ .
3. Compute limit dependencies over a grid with only the nodes needed to evaluate said limit.

### Satisfying criteria 1

The first criteria can be satisfied by adding an if condition (focus on line 16) to the compute functions of quantities which change form at the axis.

```
1 @register_compute_fun(  
2     name="B0",  
3     label="\\partial_{\\rho} \\psi / \\sqrt{g}",  
4     units="T \\cdot m^{-1}",  
5     units_long="Tesla / meter",  
6     description="",  
7     dim=1,  
8     params=[],  
9     transforms={},  
10    profiles=[],  
11    coordinates="rtz",  
12    data=["psi_r", "sqrt(g)"] + ["psi_rr", "sqrt(g)_r"],  
13 )  
14 def _B0(params, transforms, profiles, data, **kwargs):  
15     data["B0"] = data["psi_r"] / data["sqrt(g)"]  
16     if transforms["grid"].axis.size:  
17         limit = data["psi_rr"] / data["sqrt(g)_r"]  
18         data["B0"] = put(  
19             data["B0"], transforms["grid"].axis, limit[transforms["grid"].axis]  
20         )  
21     return data
```

Can probably make this logic prettier via a decorator function.

### Satisfying criteria 2

In the above example, `sqrt(g)_r` is a limit dependency (only needed for limit computations), while `sqrt(g)` is a full dependency (needed for normal computation). Notice the above code does not differentiate between limit dependencies

and full dependencies. This means criteria 2 is not satisfied because all the limit dependencies are computed even if the grid does not have a node at  $\rho = 0$ . In any implementation, to avoid computing dependencies of a quantity that are only needed for its limit, we need to mark dependencies as such in `data_index`. For each quantity, we will divide that quantity's dependencies into full dependencies and limit dependencies when defining the compute functions for each quantity. Then the second criteria can be satisfied by adding the following code

```

1 if transforms["grid"].axis and not has_limit_dependencies(name, params,
2     transforms, profiles, data):
3     data = _compute(
4         data_index[name]["limit_dependencies"]["data"],
5         params=params,
6         transforms=transforms,
7         profiles=profiles,
8         data=data,
9         **kwargs,
10    )

```

after line 16 of

```

1 def _compute(names, params, transforms, profiles, data=None, **kwargs):
2     """Same as above but without checking inputs for faster recursion."""
3     for name in names:
4         if name in data:
5             # don't compute something that's already been computed
6             continue
7         if not has_dependencies(name, params, transforms, profiles, data):
8             # then compute the missing dependencies first
9             data = _compute(
10                data_index[name]["dependencies"]["data"],
11                params=params,
12                transforms=transforms,
13                profiles=profiles,
14                data=data,
15                **kwargs,
16            )
17         data = data_index[name]["fun"](params, transforms, profiles, data, **
18            kwargs)
19     return data

```

Listing 1: Currently used code to recursively build dependencies



## Satisfying criteria 3

To compute the limit of B0, we only need to compute `sqrt(g)_r` at  $\rho = 0$  using a grid like

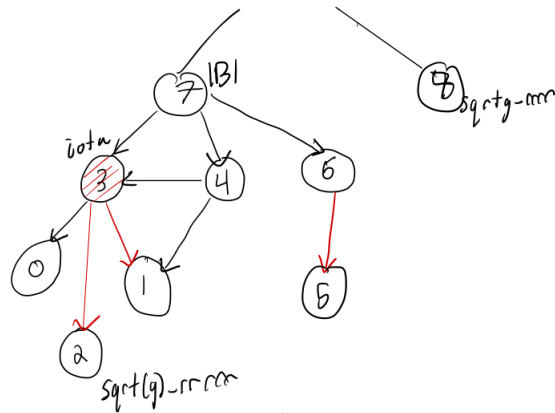
```

1 just_axis_grid = LinearGrid(
2     rho=0,
3     theta=transforms["grid"].unique_theta,
4     zeta=transforms["grid"].unique_zeta,
5     NFP=transforms["grid"].NFP,
6     sym=transforms["grid"].sym,
7 )

```

Key  
 limit dependence \      limit changes formula   
 full dependence \      limit doesn't change formula 

eq.compute(|B|, sqrt(q)-rrr, grid-with-axis)



To compute limit dependencies on this smaller grid while still computing full dependencies on a full grid, we need to pick either the greedy or lazy approach below.

- The greedy approach traverses the dependency DAG twice. The first time it marks all dependencies which are only limit dependencies. Denote the set of full dependencies  $F$  and the set of all limit dependencies  $L$ . The marked quantities are  $A \stackrel{\text{def}}{=} F \setminus L$ . Then traverse the DAG again, now computing all the dependencies. If the dependency to compute is in  $A$ , compute it on a grid with just  $\rho = 0$  axis.
- The lazy approach traverses the dependency DAG once. However, the limit dependencies will be recomputed over the full grid if it turns out that that quantity is a full dependency for some other quantity. If a limit dependency is already computed on the full grid, just use that. Recurse down full dependencies first to reduce redundant computations.

I refer to full dependencies as black and limit dependencies as red to match up with the links in this DAG.

```

1 data = eq.compute(["iota", "B", "psi"], grid_with_axis)

3 def desc.compute.utils.compute(quantities, ...):
4     pre-processing

6     --- initial recursion call ---
7     # tracks names of computed red link dependencies
8     red_link_set = set()
9     # same thing as always
10    data = dict()
11    for each quantity in quantities:
12        if quantity in red_link_set:
13            # mark to recompute on full grid
14            del data[quantity]
15            red_link_set.remove(quantity)
16        data = _compute(quantity, red_link_set, data, ...)
17    return data

19 def desc.compute.utils._compute(quantity, ...):
20     if quantity in data:
21         # don't compute something that's already been computed
22         return data

24     for each dependency of quantity:
25         if dependency is black and in red_link_set:
26             # mark to recompute on full grid
27             del data[dependency]
28             red_link_set.remove(dependency)
29         if dependency not in data:
30             # then compute this missing dependency
31             if dependency is black:
32                 data = _compute(dependency, full_grid, data)
33             else:
34                 # dependency of quantity's limit only
35                 red_link_set.add(dependency)
36                 data = _compute(dependency, just_axis_grid, data)
37     # now compute quantity
38     data = data_index[quantity]["fun"](data, ...)
39     if quantity in red_link_set:
40         # good spot to make broadcastable with full grid, but we don't need to.
41         pass
42     return data

```

Listing 2: Lazy approach

As hinted above, we can avoid broadcasting issues between quantities computed on the two different grids by selecting a better choice for the just axis grid as follows.

```

1 axis_mask = full_grid.nodes[:, 0] == 0
2 just_axis_grid = Grid(
3     nodes=full_grid.nodes[axis_mask],
4     spacing=full_grid.spacing[axis_mask],
5     sort=False,
6     # other flags to not touch the grid
7 )

```

Then, the example code in criteria 1 *simplifies* to:

```

1 @register_compute_fun(
2     name="B0",
3     label="\\partial_{{rho}} \\psi / \\sqrt{g}",
4     units="T \\cdot m^{-1}",
5     units_long="Tesla / meter",
6     description="",
7     dim=1,
8     params=[],
9     transforms={},
10    profiles=[],
11    coordinates="rtz",
12    data=["psi_r", "sqrt(g)"] + ["psi_rr", "sqrt(g)_r"],
13 )
14 def _B0(params, transforms, profiles, data, **kwargs):
15     data["B0"] = data["psi_r"] / data["sqrt(g)"]
16     if transforms["grid"].axis.size:
17         limit = data["psi_rr"] / data["sqrt(g)_r"]
18         data["B0"] = put(data["B0"], transforms["grid"].axis, limit)
19     return data

```

If the limit depended on full grid quantities as well, we would simply grab their relevant indices with `quantity[transforms["grid"].axis]` on those quantities to compute the result in line 17 above.