fv3gfs-util Documentation

Release 0.6.0

Vulcan Technologies, LLC

Feb 04, 2022

CONTENTS:

Warning: fv3gfs-util has been renamed to pace-util, and can now be found in the Pace Github repository.

fv3gfs-util is a toolkit for building Python weather and climate models. Its features can seem disjoint, which is by design - you can choose which functionality you want to use and leave the rest. It is currently used to contain pure Python utilities shared by fv3gfs-wrapper and fv3core. As the number of features increases, we may move its functionality into separate packages to reduce the dependency stack.

Some broad categories of features are:

- *fv3gfs.util.Quantity*, the data type used by fv3gfs-util described in the section on *State*
- *Communication* objects used for MPI
- Utility functions useful in weather and climate models, described in *Utilities*

CHAPTER

ONE

README

fv3gs-util has moved!

Please visit [pace](https://github.com/ai2cm/pace/tree/main/fv3gfs-util) to follow the latest development.

> This repository is no longer maintained

1.1 Installation

1.1.1 Stable release

There is no stable release. This is alpha research software: use at your own risk!

1.1.2 From sources

fv3gfs-util can be installed from Github using:

```
$ pip install git+https://github.com/VulcanClimateModeling/fv3gfs-util.git
```

The sources for fv3gfs-util can be downloaded from the Github repo. To develop fv3gfs-util, you can clone the public repository:

```
$ git clone git://github.com/VulcanClimateModeling/fv3gfs-util
```

Once you have a copy of the source, you can install it in develop mode with:

```
$ pip install -r ./fv3gfs-util/requirements.txt -c ./fv3gfs-util/constraints.txt -e␣
↪fv3gfs-util
```

The *-e* flag will set up the directory so that python uses the local folder including any modifications, instead of copying the sources to an installation directory. This is very useful for development. The *-r requirements.txt* will install extra packages useful for test, lint & other development requirements. The *-r requirements_gpu.txt* will install required packages for GPU compatibility. The *-c constraints.txt* is optional, but will ensure the package versions you use are ones we have tested against.

1.2 State

1.2.1 Quantity

Data in fv3gfs-util is managed using a container type called *fv3gfs.util.Quantity*. This stores metadata such as dimensions and units (in `quantity.dims` and `quantity.units`), and manages the "computational domain" of the data.

When running a model on multiple processors ("ranks"), such as in a cubed sphere configuration, each process is responsible for a subset of the domain, called its "compute domain" or "computational domain". Arrays may contain additional data in a "halo" of "ghost cells" which hold data from another rank's compute domain to be used as inputs for the local rank. This data needs to be periodically retrieved from nearby ranks, as the local rank cannot compute the new values outside of its compute domain.

A 3-by-3 array with one set of halo points would look something like:

```
x x x x x
x 0 0 0 x
x 0 0 0 x
x 0 0 0 x
x x x x x
```

where *0* values represent the compute domain, and *x* represents points in the halo. If you are interested in learning more, look up the "Ghost Cell Pattern" or "Halo Exchange".

Depending on optimization choices, it may also make sense to include filler data which serves only to align the computational domain into blocks within memory.

If all of that sounded confusing, we agree! That's why *fv3gfs.util.Quantity* abstracts away as much of this information as possible. If you perform indexing on the `view` attribute of quantity, the index will be applied within the computational domain:

```
quantity.view[:] = 0.  # set all data this rank is responsible for to 0
quantity.view[1:-1, :] = 1.0  # set data not on the first dimension edge to 1
array = quantity.view[:]  # gives an array accessing just the compute domain
new_array = np.copy(quantity.view[:])  # gives a copy of the compute domain
```

If you want to access data in ghost cells, instead of `.view` you should access `.data`, which is the underlying `ndarray`-like object used by the `Quantity`:

```
quantity.data[:] = 0.  # set all data this rank has, including ghost cells, to zero
quantity.data[quantity.origin[0]-3:quantity.origin[0]] == 1.  # set the left three ghost␣
↪cells to 1
array = quantity.data[quantity.origin[0]:quantity.origin[0]+quantity.extent[0]]  # same␣
↪as quantity.view[:] for a 1D quantity
```

`data` may be a numpy array or a cupy array. Both provide the same interface and can be used identically. If you would like to use the appropriate "numpy" package to manipulate your data, you can use `quantity.np`. For example, the following will give you the mean of your array, regardless of whether the data is on CPU or GPU, and regardless of whether halo values are present:

```
quantity.np.mean(quantity.view[:])
```

1.3 Communication

As mentioned when discussing *State*, each process or "rank" on a cubed sphere is responsible for a subset of the cubed sphere grid. In order to operate, the model needs to know how to partition that cubed sphere into parts for each rank, and to be able to communicate data between those ranks.

Partitioning is managed by so-called "Partitioner" objects. The *fv3gfs.util.CubedSpherePartitioner* manages the entire cubed sphere, while the *fv3gfs.util.TilePartitioner* manages one of the six faces of the cube, or a region on one of those faces. For communication, we similarly have *fv3gfs.util.CubedSphereCommunicator* and *fv3gfs.util.TileCommunicator*. Please see their API documentation for an up-to-date list of current communications routines.

1.4 Utilities

1.4.1 Nudging

Nudging functionality is provided by *fv3gfs.util.apply_nudging()* and *fv3gfs.util.get_nudging_tendencies()*. The nudging tendencies can be stored to disk by the user, for example using a *fv3gfs.util.ZarrMonitor*. A runfile using this functionality can be found in the *examples* directory.

1.4.2 Diagnostic IO

State can be persisted to disk using either *fv3gfs.util.write_state()* (described below) or *fv3gfs.util.ZarrMonitor*. The latter will coordinate between ranks to write state to a unified Zarr store. Initializing it requires passing grid information. This can be done directly from the namelist in a configuration dictionary like so:

```
import fv3gfs.util
from mpi4py import MPI
import yaml

with open('fv3config.yml', 'r') as f:
    config = yaml.safe_load(f)
partitioner = fv3gfs.util.TilePartitioner.from_namelist(config['namelist'])
```

Alternatively, the grid information can be specified manually:

```
partitioner = fv3gfs.util.TilePartitioner(
    layout=(1, 1)
)
```

Once you have a *fv3gfs.util.TilePartitioner*, the monitor can be created using any Zarr store:

```
import zarr
store = zarr.storage.DirectoryStore('output_dir')  # relative or absolute path
ZarrMonitor(partitioner, store, mode='w', mpi_comm=MPI.COMM_WORLD)
```

Note this can be used with any directory store available in `zarr`.

1.4.3 Saving state to disk

Sometimes you may want to write out model state to disk so that you can restart the model from this state later. We provide a python-centric method for saving out and loading model state. *fv3gfs.util.read_state()* saves the state on the current rank to a file on disk, while *fv3gfs.util.write_state()* writes the rank's state to disk. Make sure you use different filenames for each rank!

1.4.4 Loading Fortran Restarts

A function *fv3gfs.util.open_restart()* is available to load restart files that have been output by the Fortran FV3GFS model. This routine will handle loading the data on a single processor per tile and then distribute the data to other processes on the same tile. This may cause out-of-memory errors, which can be mitigated in a couple different ways through changes to the code base (e.g. loading a subset of the variables or levels at a time before distributing across ranks).

1.5 API

class fv3gfs.util.**Buffer**(*key: Tuple[Callable, Iterable[int], type]*, *array: numpy.ndarray*)

Bases: object

A buffer cached by default.

_key: key into cache storage to allow easy re-caching array: ndarray allocated

array: *numpy.ndarray*

assign_from(*source_array: numpy.ndarray*, *buffer_slice: numpy.lib.index_tricks.IndexExpression = (slice(None, None, None),)*)

Assign source_array to internal array.

Parameters **source_array** – source ndarray

assign_to(*destination_array: numpy.ndarray*, *buffer_slice: numpy.lib.index_tricks.IndexExpression = (slice(None, None, None),)*, *buffer_reshape: Optional[numpy.lib.index_tricks.IndexExpression] = None*)

Assign internal array to destination_array.

Parameters **destination_array** – target ndarray

finalize_memory_transfer()

Finalize any memory transfer

classmethod pop_from_cache(*allocator: fv3gfs.util.types.Allocator*, *shape: Iterable[int]*, *dtype: type*) → *fv3gfs.util.buffer.Buffer*

Retrieve or insert then retrieve of buffer from cache.

Parameters

- **allocator** – used to allocate memory
- **shape** – shape of array
- **dtype** – type of array elements

Returns a buffer wrapping an allocated array

static push_to_cache(*buffer:* fv3gfs.util.buffer.Buffer)

Push the buffer back into the cache.

Parameters **buffer** – buffer to push back in cache, using internal key

class fv3gfs.util.**Communicator**(*comm*, *partitioner*, *force_cpu: bool = False*)
Bases: object

gather(*send_quantity:* fv3gfs.util.quantity.Quantity, *recv_quantity: Optional[*fv3gfs.util.quantity.Quantity*] = None*) → Optional[*fv3gfs.util.quantity.Quantity*]
Transfer subtile regions of a full-tile quantity from each rank to the tile root rank.

Parameters

- **send_quantity** – quantity to send
- **recv_quantity** – if provided, assign received data into this Quantity (only used on the tile root rank)

Returns quantity if on root rank, otherwise None

Return type recv_quantity

gather_state(*send_state=None*, *recv_state=None*)
Transfer a state dictionary from subtile ranks to the tile root rank.

'time' is assumed to be the same on all ranks, and its value will be set to the value from the root rank.

Parameters

- **send_state** – the model state to be sent containing the subtile data
- **recv_state** – the pre-allocated state in which to recieve the full tile state. Only variables which are scattered will be written to.

Returns on the root rank, the state containing the entire tile

Return type recv_state

property rank: int
rank of the current process within this communicator

scatter(*send_quantity: Optional[*fv3gfs.util.quantity.Quantity*] = None*, *recv_quantity: Optional[*fv3gfs.util.quantity.Quantity*] = None*) → *fv3gfs.util.quantity.Quantity*
Transfer subtile regions of a full-tile quantity from the tile root rank to all subtiles.

Parameters

- **send_quantity** – quantity to send, only required/used on the tile root rank
- **recv_quantity** – if provided, assign received data into this Quantity.

Returns recv_quantity

scatter_state(*send_state=None*, *recv_state=None*)
Transfer a state dictionary from the tile root rank to all subtiles.

Parameters

- **send_state** – the model state to be sent containing the entire tile, required only from the root rank
- **recv_state** – the pre-allocated state in which to recieve the scattered state. Only variables which are scattered will be written to.

Returns the state corresponding to this rank's subdomain

Return type rank_state

class fv3gfs.util.**CubedSphereCommunicator**(*comm*, *partitioner: fv3gfs.util.partitioner.CubedSpherePartitioner*, *force_cpu: bool = False*, *timer: Optional[fv3gfs.util._timing.Timer] = None*)

Bases: *fv3gfs.util.communicator.Communicator*

Performs communications within a cubed sphere

property boundaries: Mapping[int, fv3gfs.util.boundary.Boundary]
boundaries of this tile with neighboring tiles

finish_halo_update(*quantity: fv3gfs.util.quantity.Quantity*, *n_points: int*)
Deprecated, do not use.

finish_vector_halo_update(*x_quantity: fv3gfs.util.quantity.Quantity*, *y_quantity: fv3gfs.util.quantity.Quantity*, *n_points: int*)
Deprecated, do not use.

get_scalar_halo_updater(*specifications: List[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*)

get_vector_halo_updater(*specifications_x: List[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*, *specifications_y: List[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*)

halo_update(*quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *n_points: int*)
Perform a halo update on a quantity or quantities

Parameters

- **quantity** – the quantity to be updated
- **n_points** – how many halo points to update, starting from the interior

partitioner: *fv3gfs.util.partitioner.CubedSpherePartitioner*

start_halo_update(*quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *n_points: int*) → *fv3gfs.util.halo_updater.HaloUpdater*
Start an asynchronous halo update on a quantity.

Parameters

- **quantity** – the quantity to be updated
- **n_points** – how many halo points to update, starting from the interior

Returns an asynchronous request object with a .wait() method

Return type request

start_synchronize_vector_interfaces(*x_quantity: fv3gfs.util.quantity.Quantity*, *y_quantity: fv3gfs.util.quantity.Quantity*) → *fv3gfs.util.communicator.HaloUpdateRequest*
Synchronize shared points at the edges of a vector interface variable.

Sends the values on the south and west edges to overwrite the values on adjacent subtiles. Vector must be defined on the Arakawa C grid.

For interface variables, the edges of the tile are computed on both ranks bordering that edge. This routine copies values across those shared edges so that both ranks have the same value for that edge. It also handles any rotation of vector quantities needed to move data across the edge.

Parameters

- **x_quantity** – the x-component quantity to be synchronized

- **y_quantity** – the y-component quantity to be synchronized

Returns an asynchronous request object with a .wait() method

Return type request

start_vector_halo_update(*x_quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *y_quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *n_points: int*) → *fv3gfs.util.halo_updater.HaloUpdater*

Start an asynchronous halo update of a horizontal vector quantity.

Assumes the x and y dimension indices are the same between the two quantities.

Parameters

- **x_quantity** – the x-component quantity to be halo updated
- **y_quantity** – the y-component quantity to be halo updated
- **n_points** – how many halo points to update, starting at the interior

Returns an asynchronous request object with a .wait() method

Return type request

synchronize_vector_interfaces(*x_quantity: fv3gfs.util.quantity.Quantity*, *y_quantity: fv3gfs.util.quantity.Quantity*)

Synchronize shared points at the edges of a vector interface variable.

Sends the values on the south and west edges to overwrite the values on adjacent subtiles. Vector must be defined on the Arakawa C grid.

For interface variables, the edges of the tile are computed on both ranks bordering that edge. This routine copies values across those shared edges so that both ranks have the same value for that edge. It also handles any rotation of vector quantities needed to move data across the edge.

Parameters

- **x_quantity** – the x-component quantity to be synchronized
- **y_quantity** – the y-component quantity to be synchronized

property tile: *fv3gfs.util.communicator.TileCommunicator*

communicator for within a tile

timer: *fv3gfs.util._timing.Timer*

vector_halo_update(*x_quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *y_quantity: Union[fv3gfs.util.quantity.Quantity, List[fv3gfs.util.quantity.Quantity]]*, *n_points: int*)

Perform a halo update of a horizontal vector quantity or quantities.

Assumes the x and y dimension indices are the same between the two quantities.

Parameters

- **x_quantity** – the x-component quantity to be halo updated
- **y_quantity** – the y-component quantity to be halo updated
- **n_points** – how many halo points to update, starting at the interior

class fv3gfs.util.**CubedSpherePartitioner**(*tile: fv3gfs.util.partitioner.TilePartitioner*)

Bases: fv3gfs.util.partitioner.Partitioner

boundary(*boundary_type: int*, *rank: int*) → Optional[fv3gfs.util.boundary.SimpleBoundary]
Returns a boundary of the requested type for a given rank, or None.

On tile corners, the boundary across that corner does not exist.

Parameters

- **boundary_type** – the type of boundary
- **rank** – the processor rank

Returns boundary

classmethod from_namelist(*namelist*)
Initialize a CubedSpherePartitioner from a Fortran namelist.

Parameters **namelist** (*dict*) – the Fortran namelist

global_extent(*rank_metadata:* fv3gfs.util.quantity.QuantityMetadata) → Tuple[int, ...]
Return the shape of a full cube representation for the given dimensions.

Parameters **metadata** – quantity metadata

Returns shape of full cube representation

Return type extent

property layout: Tuple[int, int]

subtile_extent(*cube_metadata:* fv3gfs.util.quantity.QuantityMetadata) → Tuple[int, ...]
Return the shape of a single rank representation for the given dimensions.

subtile_slice(*rank: int*, *global_dims: Sequence[str]*, *global_extent: Sequence[int]*, *overlap: bool = False*)
→ Tuple[Union[int, slice], ...]
Return the subtile slice of a given rank on an array.

Global refers to the domain being partitioned. For example, for a partitioning of a tile, the tile would be the "global" domain.

Parameters

- **rank** – the rank of the process
- **global_dims** – dimensions of the global quantity being partitioned
- **global_extent** – extent of the global quantity being partitioned
- **overlap** (*optional*) – if True, for interface variables include the part of the array shared by adjacent ranks in both ranks. If False, ensure only one of those ranks (the greater rank) is assigned the overlapping section. Default is False.

Returns

the tuple slice of the global compute domain corresponding to the subtile compute domain

Return type subtile_slice

tile_index(*rank: int*) → int
Returns the tile index of a given rank

tile_root_rank(*rank: int*) → int
Returns the lowest rank on the same tile as a given rank.

property total_ranks: int
the number of ranks on the cubed sphere

class fv3gfs.util.**GridSizer**(*nx: int*, *ny: int*, *nz: int*, *n_halo: int*, *extra_dim_lengths: Dict[str, int]*)

Bases: object

extra_dim_lengths: Dict[str, int]
lengths of any non-x/y/z dimensions, such as land or radiation dimensions

get_extent(*dims: Sequence[str]*) → Tuple[int, ...]

get_origin(*dims: Sequence[str]*) → Tuple[int, ...]

get_shape(*dims: Sequence[str]*) → Tuple[int, ...]

n_halo: int
number of horizontal halo points for produced arrays

nx: int
length of the x compute dimension for produced arrays

ny: int
length of the y compute dimension for produced arrays

nz: int
length of the z compute dimension for produced arrays

class fv3gfs.util.**HaloUpdateRequest**(*send_data: List[Tuple[fv3gfs.util.types.AsyncRequest, fv3gfs.util.buffer.Buffer]]*, *recv_data: List[Tuple[fv3gfs.util.types.AsyncRequest, fv3gfs.util.buffer.Buffer, numpy.ndarray]]*, *timer: Optional[fv3gfs.util._timing.Timer] = None*)

Bases: object

Asynchronous request object for halo updates.

wait()
Wait & unpack data into destination buffers Clean up by inserting back all buffers back in cache for potential reuse

class fv3gfs.util.**HaloUpdater**(*comm: Communicator*, *tag: int*, *transformers: Dict[int, fv3gfs.util.halo_data_transformer.HaloDataTransformer]*, *timer: fv3gfs.util._timing.Timer*)

Bases: object

Exchange halo information between ranks.

The class is responsible for the entire exchange and uses the __init__ to precompute the maximum of information to have minimum overhead at runtime. Therefore it should be cached for early and re-used at runtime.

- from_scalar_specifications/from_vector_specifications are used to create an HaloUpdater from a list of memory specifications
- update and start/wait trigger the halo exchange
- the class creates a "pattern" of exchange that can fit any memory given to do/start
- temporary references to the Quanitites are held between start and wait

force_finalize_on_wait()
HaloDataTransformer are finalized after a wait call

This is a temporary fix. See DSL-816 which will remove self._finalize_on_wait.

classmethod from_scalar_specifications(*comm: Communicator*, *numpy_like_module: fv3gfs.util.types.NumpyModule*, *specifications: Iterable[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*, *boundaries: Iterable[fv3gfs.util.boundary.Boundary]*, *tag: int*, *optional_timer: Optional[fv3gfs.util._timing.Timer] = None*) → *HaloUpdater*

Create/retrieve as many packed buffer as needed and queue the slices to exchange.

Parameters

- **comm** – communicator to post network messages
- **numpy_like_module** – module implementing numpy API
- **specifications** – data specifications to exchange, including number of halo points
- **boundaries** – informations on the exchange boundaries.
- **tag** – network tag (to differentiate messaging) for this node.
- **optional_timer** – timing of operations.

Returns HaloUpdater ready to exchange data.

classmethod from_vector_specifications(*comm: Communicator*, *numpy_like_module: fv3gfs.util.types.NumpyModule*, *specifications_x: Iterable[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*, *specifications_y: Iterable[fv3gfs.util.halo_data_transformer.QuantityHaloSpec]*, *boundaries: Iterable[fv3gfs.util.boundary.Boundary]*, *tag: int*, *optional_timer: Optional[fv3gfs.util._timing.Timer] = None*) → *HaloUpdater*

Create/retrieve as many packed buffer as needed and queue the slices to exchange.

Parameters

- **comm** – communicator to post network messages
- **numpy_like_module** – module implementing numpy API
- **specifications_x** – specifications to exchange along the x axis. Length must match y specifications.
- **specifications_y** – specifications to exchange along the y axis. Length must match x specifications.
- **boundaries** – informations on the exchange boundaries.
- **tag** – network tag (to differentiate messaging) for this node.
- **optional_timer** – timing of operations.

Returns HaloUpdater ready to exchange data.

start(*quantities_x: List[fv3gfs.util.quantity.Quantity]*, *quantities_y: Optional[List[fv3gfs.util.quantity.Quantity]] = None*)

Start data exchange.

update(*quantities_x: List[fv3gfs.util.quantity.Quantity]*, *quantities_y: Optional[List[fv3gfs.util.quantity.Quantity]] = None*)

Exhange the data and blocks until finished.

wait()

Finalize data exchange.

exception fv3gfs.util.**InvalidQuantityError**
Bases: Exception

class fv3gfs.util.**NullTimer**
Bases: *fv3gfs.util._timing.Timer*

A Timer class which does not actually accumulate timings.

Meant to be used in place of an optional timer.

enable()
Enable the Timer.

property enabled: bool
Indicates whether the timer is currently enabled.

exception fv3gfs.util.**OutOfBoundsError**
Bases: ValueError

class fv3gfs.util.**Quantity**(*data*, *dims: Sequence[str]*, *units: str*, *origin: Optional[Sequence[int]] = None*, *extent: Optional[Sequence[int]] = None*, *gt4py_backend: Optional[str] = None*)
Bases: object

Data container for physical quantities.

property attrs: dict

property data: numpy.ndarray
the underlying array of data

property data_array: xarray.core.dataarray.DataArray

property dims: Tuple[str, ...]
names of each dimension

property extent: Tuple[int, ...]
the shape of the computational domain

classmethod from_data_array(*data_array: xarray.core.dataarray.DataArray*, *origin: Optional[Sequence[int]] = None*, *extent: Optional[Sequence[int]] = None*, *gt4py_backend: Optional[str] = None*) → *fv3gfs.util.quantity.Quantity*
Initialize a Quantity from an xarray.DataArray.

Parameters

- **data_array** –
- **origin** – first point in data within the computational domain
- **extent** – number of points along each axis within the computational domain
- **gt4py_backend** – backend to use for gt4py storages, if not given this will be derived from a Storage if given as the data argument, otherwise the storage attribute is disabled and will raise an exception

property gt4py_backend: Optional[str]

property metadata: *fv3gfs.util.quantity.QuantityMetadata*

property np: fv3gfs.util.types.NumpyModule

property origin: Tuple[int, ...]
the start of the computational domain

sel(***kwargs: Union[slice, int]*) → numpy.ndarray
Convenience method to perform indexing on *view* using dimension names without knowing dimension order.

Parameters ****kwargs** – slice/index to retrieve for a given dimension name

Returns

an ndarray-like selection of the given indices on *self.view*

Return type view_selection

property storage
A gt4py storage representing the data in this Quantity.

Will raise TypeError if the gt4py backend was not specified when initializing this object, either by providing a Storage for data or explicitly specifying a backend.

transpose(*target_dims: Sequence[Union[str, Iterable[str]]]*) → *fv3gfs.util.quantity.Quantity*
Change the dimension order of this Quantity.

If you know you are working with cell-centered variables, you can do:

```
>>> from fv3gfs.util import X_DIM, Y_DIM, Z_DIM
>>> transposed_quantity = quantity.transpose([X_DIM, Y_DIM, Z_DIM])
```

To support re-ordering without checking whether quantities are on cell centers or interfaces, the API supports giving a list of dimension names for dimensions. For example, to re-order to X-Y-Z dimensions regardless of the grid the variable is on, one could do:

```
>>> from fv3gfs.util import X_DIMS, Y_DIMS, Z_DIMS
>>> transposed_quantity = quantity.transpose([X_DIMS, Y_DIMS, Z_DIMS])
```

Parameters **target_dims** – a list of output dimensions. Instead of a single dimension name, an iterable of dimensions can be used instead for any entries. For example, you may want to use fv3gfs.util.X_DIMS to place an x-dimension without knowing whether it is on cell centers or interfaces.

Returns Quantity with the requested output dimension order

Return type transposed

Raises **ValueError** – if any of the target dimensions do not exist on this Quantity, or if this Quantity contains multiple values from an iterable entry

property units: str
units of the quantity

property values: numpy.ndarray

property view: fv3gfs.util.quantity.BoundedArrayView
a view into the computational domain of the underlying data

class fv3gfs.util.**QuantityFactory**(*sizer:* fv3gfs.util.initialization.sizer.SubtileGridSizer, *numpy*)
Bases: object

empty(*dims: typing.Sequence[str], units: str, dtype: type* = <*class 'float'*>)

classmethod from_backend(*sizer:* fv3gfs.util.initialization.sizer.SubtileGridSizer, *backend: str*)
Initialize a QuantityFactory to use a specific gt4py backend.

Parameters

- **sizer** – object which determines array sizes
- **backend** – gt4py backend

ones(*dims: typing.Sequence[str], units: str, dtype: type = <class 'float'>*)

zeros(*dims: typing.Sequence[str], units: str, dtype: type = <class 'float'>*)

class fv3gfs.util.**QuantityHaloSpec**(*n_points: int*, *strides: Tuple[int]*, *itemsize: int*, *shape: Tuple[int]*, *origin: Tuple[int, ...]*, *extent: Tuple[int, ...]*, *dims: Tuple[str, ...]*, *numpy_module: fv3gfs.util.types.NumpyModule*, *dtype: Any*)

Bases: object

Describe the memory to be exchanged, including size of the halo.

dims: Tuple[str, ...]

dtype: Any

extent: Tuple[int, ...]

itemsize: int

n_points: int

numpy_module: fv3gfs.util.types.NumpyModule

origin: Tuple[int, ...]

shape: Tuple[int]

strides: Tuple[int]

class fv3gfs.util.**QuantityMetadata**(*origin: Tuple[int, ...]*, *extent: Tuple[int, ...]*, *dims: Tuple[str, ...]*, *units: str*, *data_type: type*, *dtype: type*, *gt4py_backend: Union[str, NoneType] = None*)

Bases: object

data_type: type
ndarray-like type used to store the data

property dim_lengths: Dict[str, int]
mapping of dimension names to their lengths

dims: Tuple[str, ...]
names of each dimension

dtype: type
dtype of the data in the ndarray-like object

extent: Tuple[int, ...]
the shape of the computational domain

gt4py_backend: Optional[str] = None
backend to use for gt4py storages

property np: fv3gfs.util.types.NumpyModule
numpy-like module used to interact with the data

origin: Tuple[int, ...]
the start of the computational domain

units: str
units of the quantity

class fv3gfs.util.**SubtileGridSizer**(*nx: int*, *ny: int*, *nz: int*, *n_halo: int*, *extra_dim_lengths: Dict[str, int]*)

Bases: *fv3gfs.util.initialization.sizer.GridSizer*

property dim_extents: Dict[str, int]

extra_dim_lengths: Dict[str, int]
lengths of any non-x/y/z dimensions, such as land or radiation dimensions

classmethod from_namelist(*namelist: dict*, *tile_partitioner: Optional[fv3gfs.util.partitioner.TilePartitioner] = None*, *tile_rank: int = 0*)
Create a SubtileGridSizer from a Fortran namelist.

Parameters

- **namelist** – A namelist for the fv3gfs fortran model
- **tile_partitioner** (*optional*) – a partitioner to use for segmenting the tile. By default, a TilePartitioner is used.
- **tile_rank** (*optional*) – current rank on tile. Default is 0. Only matters if different ranks have different domain shapes. If tile_partitioner is a TilePartitioner, this argument does not matter.

classmethod from_tile_params(*nx_tile: int*, *ny_tile: int*, *nz: int*, *n_halo: int*, *extra_dim_lengths: Dict[str, int]*, *layout: Tuple[int, int]*, *tile_partitioner: Optional[fv3gfs.util.partitioner.TilePartitioner] = None*, *tile_rank: int = 0*)
Create a SubtileGridSizer from parameters about the full tile.

Parameters

- **nx_tile** – number of x cell centers on the tile
- **ny_tile** – number of y cell centers on the tile
- **nz** – number of vertical levels
- **n_halo** – number of halo points
- **extra_dim_lengths** – lengths of any non-x/y/z dimensions, such as land or radiation dimensions
- **layout** – (y, x) number of ranks along tile edges
- **tile_partitioner** (*optional*) – partitioner object for the tile. By default, a TilePartitioner is created with the given layout
- **tile_rank** (*optional*) – rank of this subtile.

get_extent(*dims: Iterable[str]*) → Tuple[int, ...]

get_origin(*dims: Iterable[str]*) → Tuple[int, ...]

get_shape(*dims: Iterable[str]*) → Tuple[int, ...]

n_halo: int
number of horizontal halo points for produced arrays

nx: int
length of the x compute dimension for produced arrays

ny: int
length of the y compute dimension for produced arrays

nz: int
length of the z compute dimension for produced arrays

class fv3gfs.util.**TileCommunicator**(*comm*, *partitioner: fv3gfs.util.partitioner.TilePartitioner*, *force_cpu: bool = False*)

Bases: *fv3gfs.util.communicator.Communicator*

Performs communications within a single tile or region of a tile

class fv3gfs.util.**TilePartitioner**(*layout: Tuple[int, int]*)

Bases: fv3gfs.util.partitioner.Partitioner

boundary(*boundary_type: int*, *rank: int*) → Optional[fv3gfs.util.boundary.SimpleBoundary]

Returns a boundary of the requested type for a given rank.

Target ranks will be on the same tile as the given rank, wrapping around as in a doubly-periodic boundary condition.

Parameters

- **boundary_type** – the type of boundary
- **rank** – the processor rank

Returns boundary

fliplr_rank(*rank: int*) → int

classmethod from_namelist(*namelist*)

Initialize a TilePartitioner from a Fortran namelist.

Parameters **namelist** (*dict*) – the Fortran namelist

global_extent(*rank_metadata: fv3gfs.util.quantity.QuantityMetadata*) → Tuple[int, ...]

Return the shape of a full tile representation for the given dimensions.

Parameters **metadata** – quantity metadata

Returns shape of full tile representation

Return type extent

on_tile_bottom(*rank: int*) → bool

on_tile_left(*rank: int*) → bool

on_tile_right(*rank: int*) → bool

on_tile_top(*rank: int*) → bool

rotate_rank(*rank: int*, *n_clockwise_rotations: int*) → int

subtile_extent(*global_metadata: fv3gfs.util.quantity.QuantityMetadata*) → Tuple[int, ...]

Return the shape of a single rank representation for the given dimensions.

subtile_index(*rank: int*) → Tuple[int, int]

Return the (y, x) subtile position of a given rank as an integer number of subtiles.

subtile_slice(*rank: int*, *global_dims: Sequence[str]*, *global_extent: Sequence[int]*, *overlap: bool = False*) → Tuple[slice, ...]

Return the subtile slice of a given rank on an array.

Global refers to the domain being partitioned. For example, for a partitioning of a tile, the tile would be the "global" domain.

Parameters

- **rank** – the rank of the process
- **global_dims** – dimensions of the global quantity being partitioned

- **global_extent** – extent of the global quantity being partitioned
- **overlap** (*optional*) – if True, for interface variables include the part of the array shared by adjacent ranks in both ranks. If False, ensure only one of those ranks (the greater rank) is assigned the overlapping section. Default is False.

Returns

the slice of the global compute domain corresponding to the subtile compute domain

Return type subtile_slice

property total_ranks: int

class fv3gfs.util.**Timer**

Bases: object

Class to accumulate timings for named operations.

clock(*name: str*)

Context manager to produce timings of operations.

Parameters **name** – the name of the operation being timed

Example

The context manager times operations that happen within its context. The following would time a time.sleep operation:

```
>>> import time
>>> from fv3gfs.util import Timer
>>> timer = Timer()
>>> with timer.clock("sleep"):
...     time.sleep(1)
...
>>> timer.times
{'sleep': 1.0032463260000029}
```

disable()

Disable the Timer.

enable()

Enable the Timer.

property enabled: bool

Indicates whether the timer is currently enabled.

property hits: Mapping[str, int]

accumulated hit counts for each operation name

reset()

Remove all accumulated timings.

start(*name: str*)

Start timing a given named operation.

stop(*name: str*)

Stop timing a given named operation, add the time elapsed to accumulated timing and increase the hit count.

property times: Mapping[str, float]

accumulated timings for each operation name

exception fv3gfs.util.**UnitsError**
Bases: Exception

class fv3gfs.util.**ZarrMonitor**(*store: typing.Union[str, zarr.storage.MutableMapping], partitioner: fv3gfs.util.partitioner.CubedSpherePartitioner, mode: str = 'w', mpi_comm=<fv3gfs.util.zarr_monitor.DummyComm object>*)
Bases: object

sympl.Monitor-style object for storing model state dictionaries in a Zarr store.

store(*state: dict*) → None
Append the model state dictionary to the zarr store.

Requires the state contain the same quantities with the same metadata as the first time this is called. Quantities are stored with dimensions [time, rank] followed by the dimensions included in any one state snapshot. The one exception is "time" which is stored with dimensions [time].

fv3gfs.util.**apply_nudging**(*state*, *reference_state*, *nudging_timescales: Mapping[str, datetime.timedelta]*, *timestep: datetime.timedelta*)
Nudge the given state towards the reference state according to the provided nudging timescales.

Nudging is applied to the state in-place.

Parameters

- **state** (*dict*) – A state dictionary.
- **reference_state** (*dict*) – A reference state dictionary.
- **nudging_timescales** (*dict*) – A dictionary whose keys are standard names and values are timedelta objects indicating the relaxation timescale for that variable.
- **timestep** (*timedelta*) – length of the timestep

Returns

A dictionary whose keys are standard names and values are Quantity objects indicating the nudging tendency of that standard name.

Return type nudging_tendencies (dict)

fv3gfs.util.**array_buffer**(*allocator: fv3gfs.util.types.Allocator*, *shape: Iterable[int]*, *dtype: type*) → Generator[*fv3gfs.util.buffer.Buffer*, *fv3gfs.util.buffer.Buffer*, None]
A context manager providing a contiguous array, which may be re-used between calls.

Parameters

- **allocator** – a function with the same signature as numpy.zeros which returns an ndarray
- **shape** – the shape of the desired array
- **dtype** – the dtype of the desired array

Yields *buffer_array* –

an ndarray created according to the specification in the args. May be retained and re-used in subsequent calls.

fv3gfs.util.**capture_stream**(*stream*)

fv3gfs.util.**datetime64_to_datetime**(*dt64: numpy.datetime64*) → datetime.datetime

fv3gfs.util.**ensure_equal_units**(*units1: str*, *units2: str*) → None

fv3gfs.util.**fill_scalar_corners**(*quantity:* fv3gfs.util.quantity.Quantity, *direction: typing_extensions.Literal[x, y]*, *tile_partitioner:* fv3gfs.util.partitioner.TilePartitioner, *rank: int*, *n_halo: int*)

At the corners of tile faces, copy data from halo edges into halo corners to allow stencils to be translated along those edges in a computationally-relevant way.

The quantity is modified in-place.

Parameters

- **quantity** – the quantity to modify, whose first two dimensions must be along the x and y directions, respectively
- **direction** – the direction along which we want to enable stencils to compute. For example, calling with "x" would allow a stencil with length > 1 along the x-direction to be convolved with Quantity. Note it is not possible to use corner filling to convolve with stencils having length > 1 along both x and y dimensions.
- **tile_partitioner** – object to determine tile positions of ranks
- **rank** – rank on which the quantity exists
- **n_halo** – number of halo points to fill

fv3gfs.util.**get_nudging_tendencies**(*state*, *reference_state*, *nudging_timescales: Mapping[str, datetime.timedelta]*)

Return the nudging tendency of the given state towards the reference state according to the provided nudging timescales.

Parameters

- **state** (*dict*) – A state dictionary.
- **reference_state** (*dict*) – A reference state dictionary.
- **nudging_timescales** (*dict*) – A dictionary whose keys are standard names and values are timedelta objects indicating the relaxation timescale for that variable.

Returns

A dictionary whose keys are standard names and values are Quantity objects indicating the nudging tendency of that standard name.

Return type nudging_tendencies (dict)

fv3gfs.util.**get_tile_index**(*rank: int*, *total_ranks: int*) → int

Returns the zero-indexed tile number, given a rank and total number of ranks.

fv3gfs.util.**get_tile_number**(*tile_rank: int*, *total_ranks: int*) → int

Deprecated: use get_tile_index.

Returns the tile number for a given rank and total number of ranks.

fv3gfs.util.**open_restart**(*dirname: str*, *communicator:* fv3gfs.util.communicator.CubedSphereCommunicator, *label: str = ''*, *only_names: Optional[Iterable[str]] = None*, *to_state: Optional[dict] = None*, *tracer_properties: Optional[Mapping[str, Mapping[str, Union[str, Iterable[str]]]]] = None*)

Load restart files output by the Fortran model into a state dictionary.

Parameters

- **dirname** – location of restart files, can be local or remote
- **communicator** – object for communication over the cubed sphere

- **label** – prepended string on the restart files to load
- **only_names** (*optional*) – list of standard names to load
- **to_state** (*optional*) – if given, assign loaded data into pre-allocated quantities in this state dictionary

Returns model state dictionary

Return type state

fv3gfs.util.**read_state**(*filename: str*) → dict
Read a model state from a NetCDF file.

Parameters **filename** – local or remote location of the NetCDF file

Returns a model state dictionary

Return type state

fv3gfs.util.**recv_buffer**(*allocator: Callable*, *array: numpy.ndarray*, *timer: Optional[fv3gfs.util._timing.Timer] = None*) → numpy.ndarray
A context manager ensuring that array is contiguous in a context where it is being used to receive data, using a recycled buffer array and then copying the result into array if necessary.

Parameters

- **allocator** – used to allocate memory
- **array** – a possibly non-contiguous array for which to provide a buffer
- **timer** – object to accumulate timings for "unpack"

Yields *buffer_array* –

if array is non-contiguous, a contiguous buffer array which is copied into array when the context is exited. Otherwise, yields array.

fv3gfs.util.**send_buffer**(*allocator: Callable*, *array: numpy.ndarray*, *timer: Optional[fv3gfs.util._timing.Timer] = None*) → numpy.ndarray
A context manager ensuring that *array* is contiguous in a context where it is being sent as data, copying into a recycled buffer array if necessary.

Parameters

- **allocator** – used to allocate memory
- **array** – a possibly non-contiguous array for which to provide a buffer
- **timer** – object to accumulate timings for "pack"

Yields *buffer_array* –

if array is non-contiguous, a contiguous buffer array containing the data from array. Otherwise, yields array.

fv3gfs.util.**to_dataset**(*state*)

fv3gfs.util.**units_are_equal**(*units1: str*, *units2: str*) → bool

fv3gfs.util.**write_state**(*state: dict*, *filename: str*) → None
Write a model state to a NetCDF file.

Parameters

- **state** – a model state dictionary
- **filename** – local or remote location to write the NetCDF file

1.6 History

1.6.1 latest

Major changes: - Added NullTimer to use for default Timer value, it is a disabled timer which cannot be enabled (raises NotImplementedError)

Fixes: - Fixed bug where ZarrMonitor depended on dict *.items()* always returning items in the same order

1.6.2 v0.6.0

Major changes: - Use *cftime.datetime* objects to represent datetimes instead of *datetime.datetime* objects. This results in times stored in a format compatible with the fortran model, and accurate internal representation of times with the calendar specified in the *coupler_nml* namelist. - *Timer* class is added, with methods *start* and *stop*, and properties *clock* (context manager), and *times* (dictionary of accumulated timing) - *CubedSphereCommunicator* instances now have a *.timer* attribute, which accumulates times for "pack", "unpack", "Isend", and "Recv" during halo updates - make *SubtileGridSizer.from_tile_params* public API - New method *CubedSphereCommunicator.synchronize_vector_interfaces* which synchronizes edge values on interface variables which are duplicated between adjacent ranks - Added *.sel* method to corner views (e.g. *quantity.view.northeast.sel(x=0, y=1)*) to allow indexing these corner views with arbitrary dimension ordering. - Halo updates now use tagged send/recv operations, which prevents deadlocks in certain situations - Quantity.data is now guaranteed to be a numpy or cupy array matching its *.np* module, and will no longer be a gt4py Storage - Quantity accepts a *gt4py_backend* on initialize which is used to create its *.storage* if one was not used on initialize - parent MPI rank now referred to as "root" rank in variable names and documentation - Added TILE_DIM constant for tile dimension of global quantities - Added Partitioner base class implementing features necessary for scatter/gather - Moved scatter and gather from TileCommunicator to the Communicator base class, so its code can be re-used by the CubedSphereCommunicator - Implemented subtile_slice, global_extent, and subtile_extent routines on CubedSpherePartitioner necessary for scatter/gather in CubedSphereCommunicator - Renamed argument *tile_extent* and *tile_dims* to *global_extent* and *global_dims* in routines to refer generically to the tile in the case of tile scatter/gather or cube in the case of cube scatter/gather - Fixed a bug where initializing a Quantity with a numpy array and a gpu backend would give CPUStorage - raise TypeError if initializing a quantity with both a storage and a gt4py_backend argument - eagerly create storage object when initializing Quantity - make data type of quantity and storage reflect the gt4py_backend chosen, instead of being determined based on the data type being numpy/cupy

Fixes: - If *only_names* is provided to *open_restart*, it will return those fields and nothing more. Previously it would include *"time"* in the returned state even if it was not requested. - Fixed a bug where quantity.storage and quantity.data could be out of sync if the quantity was initialized using data and a gt4py backend string - Default slice for corner views when not given at all as an index (e.g. when providing one index to a 2D view) now gives the same result as providing an empty slice (:) - Fixed a bug where quantity.view could refer to a different array than quantity.data if the quantity was initialized using data and a gt4py backend string, and then quantity.storage was accessed

1.6.3 v0.5.1

- enable MPI tests on CircleCI

1.6.4 v0.5.0

Breaking changes: - *send_buffer* and *recv_buffer* are modified to take in a *callable*, which is more easily serialized than a *numpy*-like module (necessary because we serialize the arguments to re-use buffers), and allows custom specification of the initialization if zeros are needed instead of empty.

Major changes: - Added additional regional views to Quantity as attributes on Quantity.view, including *northeast*, *northwest*, *southeast*, *southwest*, and *interior* - Separated fv3util into its own repository and began tracking history separately from fv3gfs-python - Added getters and setters for additional dynamics quantities needed to call an alternative dynamical core - Added *storage* property to Quantity, implemented as short-term shortcut to .data until gt4py GDP-3 is implemented

Deprecations: - *Quantity.values* is deprecated

1.6.5 v0.4.3 (2020-05-15)

Last release of fv3util with history contained in fv3gfs-python.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

T

U

V

W

Z