Sympl Documentation

Release 0.3.2

Rodrigo Caballero

May 22, 2018

Users

sympl is an open source project aims to enable researchers and other users to write understandable, modular, accessible Earth system and planetary models in Python. It is meant to be used in combination with other packages that provide model components in order to write model scripts. Its source code can be found on GitHub.

CHAPTER 1

Documentation

1.1 What's New

1.1.1 Latest

1.1.2 v0.3.2

- Exported get_constants_string to the public API
- Added "aliases" kwarg to NetCDFMonitor, allowing the monitor to shorten variable names when writing to netCDF
- Added get_component_aliases() to get a dictionary of quantity aliases from a list of Components (used by NetCDFMonitor to shorten variable names)
- Added tests for NetCDFMonitor aliases and get_component_aliases()

Breaking changes

- tendencies in diagnostics are now named as X_tendency_from_Y, instead of tendency_of_X_due_to_Y. The idea is that it's shorter, and can easily be shortened more by aliasing "tendency" to "tend"

1.1.3 v0.3.1

- Fixed botched deployment, see v0.3.0 for the real changes

1.1.4 v0.3.0

- Modified component class checking to look at the presence of properties
- Added ScalingWrapper

- Fixed bug in TendencyInDiagnosticsWrapper where tendency_diagnostics_properties were being copied into input_properties
- Modified component class checking to look at the presence of properties attributes instead of checking type when verifying component class.
- Removed Python 3.4 from Travis CI testing
- added some more constants to default_constants related to conductivity of water in all phases and phase changes of water.
- increased the verbosity of the error output on shape mismatch in restore_data_arrays_with_properties
- corrected heat capacity of snow and ice to be floats instead of ints
- Added get_constant function as the way to retrieve constants
- Added ImplicitPrognostic as a new component type. It is like a Prognostic, but its call signature also requires that a timestep be given.
- Added TimeDifferencingWrapper, which turns an Implicit into an ImplicitPrognostic by applying first-order time differencing.
- Added set_condensible_name as a way of changing what condensible aliases (for example, density_of_solid_phase) refer to. Default is 'water'.
- Moved wrappers to their own file (out from util.py).
- Corrected str representation of Diagnostic to say Diagnostic instead of Implicit.
- Added a function reset_constants to reset the constants library to its initial state.
- Added a function datetime which accepts calendar as a keyword argument, and returns datetimes from netcdftime when non-default calendars are used. The dependency on netcdftime is optional, the other calendars just won't work if it isn't installed
- Added a reference to the built-in timedelta for convenience.

Breaking changes

- Removed default_constants from the public API, use get_constant and set_constant instead.
- Removed replace_none_with_default. Use get_constant instead.
- set_dimension_names has been removed, use set_direction_names instead.

1.1.5 0.2.1

- Fixed value of planetary radius, added specific heat of water vapor.
- Added function set_constant which provides an easy interface for setting values in the default_constants dictionary. Users can already set them manually by creating DataArray objects. This automates the DataArray creation, which should make user code cleaner.

1.1.6 0.2.0

- Added some more physical constants.
- Added readthedocs support.
- Overhaul of documentation.

- Docstrings now use numpy style instead of Google style.
- Expanded tests.
- Added function to put prognostic tendencies in diagnostic output.
- NetCDFMonitor is actually working now, and has tests.
- There are now helper functions for automatically extracting required numpy arrays with correct dimensions and units from input state dictionaries. See the note about _properties attributes in Breaking changes below.
- Added base object for testing components
- Renamed set_dimension_names to set_direction_names, set_dimension_names is now deprecated and gives a warning. add_direction_names was added to append to the dimension list instead of replacing it.

Breaking changes

- The constant `stefan_boltzmann` is now called `stefan_boltzmann_constant` to maintain consistency with other names.
- Removed add_dicts_inplace from public API
- combine_dimensions will raise exceptions in a few more cases where it should do so. Particularly, if there is an extra dimension in the arrays.
- Default out_dims is removed from combine_dimensions.
- input_properties, tendency_properties, etc. dictionaries have been added to components, which contain information about the units and dimensions required for those arrays, and can include more properties as required by individual projects. This makes it possible to extract appropriate numpy arrays from a model state in an automated fashion based on these properties, significantly reducing boilerplate code. These dictionaries need to be defined by subclasses, instead of the old "inputs", "outputs" etc. lists which are auto-generated from these new dictionaries.
- Class wrapping now works by inheritance, instead of by monkey patching methods.
- All Exception classes (e.g. SharedKeyException) have been renamed to "Error" classes (e.g. SharedKeyError) to be consistent with normal Python naming conventions

1.1.7 0.1.1 (2017-01-05)

- First release on PyPI.

1.2 Overview: Why Sympl?

Traditional atmospheric and Earth system models can be difficult to understand and modify for a number of reasons. Sympl aims to learn from the past experience of these models to accelerate research and improve accessibility.

Sympl defines a framework of Python object APIs that can be combined to create a model. This has a number of benefits:

- Objects can use code written in any language that can be called from Python, including Fortran, C, C++, Julia, Matlab, and others.
- Each object, such as a radiation parameterization, has a clearly documented interface and can be understood without looking at any other part of a model's code. Certain interfaces have been designed to force model code to self-document, such as having inputs and outputs as properties of a scheme.

- Objects can be swapped out with other compatible objects. For example, Sympl makes it trivial to change the type of time stepping used.
- Code can be re-used between different types of models. For instance, an atmospheric general circulation model, numerical weather prediction model, and large-eddy simulation could all use the same RRTM radiation object.
- Already-existing documentation for Sympl can tell your users how to configure and run your model. You will likely spend less time writing documentation, but end up with a better documented model. As long as you write docstrings, you're good to go!

Sympl also contains a number of commonly used objects, such as time steppers and NetCDF output objects.

1.2.1 So is Sympl a model?

Sympl is *not* a model itself. In particular, physical parameterizations and dynamical cores are not present in Sympl. This code instead can be found in other projects that make use of Sympl.

Sympl is meant to be a community ecosystem that allows researchers and other users to use and combine components from a number of different sources. By keeping model physics/dynamics code outside of Sympl itself, researchers can own and maintain their own models. The framework API ensures that models using Sympl are clear and accessible, and allows components from different models and packages to be used alongside one another.

1.2.2 Then where's the model?

Models created with Sympl can work differently from traditional Fortran models. A model developer makes the components of their model available. Using these components, you can write a script which acts as the model executable, but also configures the model, and calls any online analysis you want to run. Model developers may make example model scripts available which you can modify.

In a way, when you configure the model you are writing the model itself. This is reasonable in Sympl because the model run script should be accessible and readable by users with basic knowledge of programming (even users who don't know Python). By being readable, the model run script tells others clearly and precisely how you configured and ran your model.

1.2.3 The API

In a Sympl model, the model state is contained within a "state dictionary". This is a Python dictionary whose keys are strings indicating a quantity, and values are DataArrays with the values of those quantities. The one exception is "time", which is stored as a timedelta or datetime-like object, not as a DataArray. The DataArrays also contain information about the units of the quantity, and the grid it is located on. At the start of a model script, the state dictionary should be set to initial values. Code to do this may be present in other packages, or you can write this code yourself. The state and its initialization is discussed further in *Model State*.

The state dictionary is evolved by `TimeStepper` and `Implicit` objects. These types of objects take in the state and a timedelta object that indicates the time step, and return the next model state. `TimeStepper` objects do this by wrapping `Prognostic` objects, which calculate tendencies using the state dictionary. We should note that the meaning of "Implicit" in Sympl is slightly different than its traditional definition. Here an "Implicit" object is one that calculates the new state directly from the current state, or any object that requires the timestep to calculate the new state, while "Prognostic" objects are ones that calculate tendencies without using the timestep. If a `TimeStepper` or `Implicit` object needs to use multiple time steps in its calculation, it does so by storing states it was previously given until they are no longer needed.

The state is also calculated using `Diagnostic` objects which determine diagnostic quantities at the current time from the current state, returning them in a new dictionary. This type of object is particularly useful if you want to write your own online diagnostics.

The state can be stored or viewed using *Monitor* objects. These take in the model state and do something with it, such as storing it in a NetCDF file, or updating an interactive plot that is being shown to the user.

1.3 Quickstart

Here we have an example of how Sympl might be used to construct a model run script, with explanations of what's going on. Here is the full model script we will be looking at:

```
from model_package import (
    get_initial_state, Radiation, BoundaryLayer, DeepConvection,
    ImplicitDynamics)
from sympl import (
    AdamsBashforth, PlotFunctionMonitor, UpdateFrequencyWrapper,
    datetime, timedelta)

def my_plot_function(fig, state):
    ax = fig.add_subplot(1, 1, 1)
    ax.set_xlabel('longitude')
    ax.set_ylabel('latitude')
    ax.set_title('Lowest model level air temperature (K)')
    im = ax.pcolormesh(
        state['air_temperature'].to_units('degK').values[0, :, :],
        vmin=260.,
        vmax=310.)
    cbar = fig.colorbar(im)

plot_monitor = PlotFunctionMonitor(my_plot_function)

state = get_initial_state(nx=256, ny=128, nz=64)
state['time'] = datetime(2000, 1, 1)

physics_stepper = AdamsBashforth([
    UpdateFrequencyWrapper(Radiation(), timedelta(hours=2)),
    BoundaryLayer(),
    DeepConvection(),
])
implicit_dynamics = ImplicitDynamics()

timestep = timedelta(minutes=30)
while state['time'] < datetime(2010, 1, 1):
    physics_diagnostics, state_after_physics = physics_stepper(state, timestep)
    dynamics_diagnostics, next_state = implicit_dynamics(state_after_physics,␣
↪timestep)
    state.update(physics_diagnostics)
    state.update(dynamics_diagnostics)
    plot_monitor.store(state)
    next_state['time'] = state['time'] + timestep
    state = next_state
```

1.3.1 Importing Packages

At the beginning of the script we have import statements:

```
from model_package import (
    get_initial_state, Radiation, BoundaryLayer, DeepConvection,
    ImplicitDynamics)
from sympl import (
    AdamsBashforth, PlotFunctionMonitor, UpdateFrequencyWrapper,
    datetime, timedelta)
```

These grant access to the objects that will be used to construct the model, and are dependent on the model package you are using. Here, the names `model_package`, `get_initial_state`, `Radiation`, `BoundaryLayer`, `DeepConvection`, and `ImplicitDynamics` are placeholders, and do not refer to an actual existing package.

1.3.2 Defining a PlotFunctionMonitor

Here we define a plotting function, and use it to create a *Monitor* using *PlotFunctionMonitor*:

```
def my_plot_function(fig, state):
    ax = fig.add_subplot(1, 1, 1)
    ax.set_xlabel('longitude')
    ax.set_ylabel('latitude')
    ax.set_title('Lowest model level air temperature (K)')
    im = ax.pcolormesh(
        state['air_temperature'].to_units('degK').values[0, :, :],
        vmin=260.,
        vmax=310.)
    cbar = fig.colorbar(im)

plot_monitor = PlotFunctionMonitor(my_plot_function)
```

That *Monitor* will be used to produce an animated plot of the lowest model level air temperature as the model runs. Here we assume that the first axis is the vertical axis, and that the lowest level is at the lowest index, but this depends entirely on your model. The `[0, :, :]` part might be different for your model.

1.3.3 Initialize the Model State

To initialize the model, we need to create a dictionary which contains the model state. The way this is done is model-dependent. Here we assume there is a function that was defined by the *model_package* package which handles this for us:

```
state = get_initial_state(nx=256, ny=128, nz=64)
state['time'] = datetime(2000, 1, 1)
```

An initialized *state* is a dictionary whose keys are strings (like 'air_temperature') and values are *DataArray* objects, which store not only the data but also metadata like units. The one exception is the "time" quantity which is either a *datetime*-like or *timedelta*-like object. Here we are calling *sympl.datetime()* to initialize time, rather than directly creating a Python datetime. This is because *sympl.datetime()* can support a number of calendars using the *netcdftime* package, if installed, unlike the built-in *datetime* which only supports the Proleptic Gregorian calendar.

You can read more about the *state*, including *sympl.datetime()* in *Model State*.

1.3.4 Initialize Components

Now we need the objects that will process the state to move it forward in time. Those are the "components":

```
physics_stepper = AdamsBashforth([
    UpdateFrequencyWrapper(Radiation(), timedelta(hours=2)),
    BoundaryLayer(),
    DeepConvection(),
])
implicit_dynamics = ImplicitDynamics()
```

AdamsBashforth is a *TimeStepper*, which is created with a set of *Prognostic* components. The *Prognostic* components we have here are *Radiation*, *BoundaryLayer*, and *DeepConvection*. Each of these carries information about what it takes as inputs and provides as outputs, and can be called with a model state to return tendencies for a set of quantities. The *TimeStepper* uses this information to step the model state forward in time.

The `UpdateFrequencyWrapper` applied to the *Radiation* object is an object that acts like a *Prognostic* but only computes its output if at least a certain amount of model time has passed since the last time the output was computed. Otherwise, it returns the last computed output. This is commonly used in atmospheric models to avoid doing radiation calculations (which are very expensive) every timestep, but it can be applied to any Prognostic.

The `ImplicitDynamics` class is a *Implicit* object, which steps the model state forward in time in the same way that a *TimeStepper* would, but doesn't use *Prognostic* objects in doing so.

1.3.5 The Main Loop

With everything initialized, we have the part of the code where the real computation is done – the main loop:

```
timestep = timedelta(minutes=30)
while state['time'] < datetime(2010, 1, 1):
    physics_diagnostics, state_after_physics = physics_stepper(state, timestep)
    dynamics_diagnostics, next_state = implicit_dynamics(state_after_physics,␣
↪timestep)
    state.update(physics_diagnostics)
    state.update(dynamics_diagnostics)
    plot_monitor.store(state)
    next_state['time'] = state['time'] + timestep
    state = next_state
```

In the main loop, a series of component calls update the state, and the figure presented by `plot_monitor` is updated. The code is meant to be as self-explanatory as possible. It is necessary to manually set the time of the next state at the end of the loop. This is not done automatically by *TimeStepper* and *Implicit* objects, because in many models you may want to update the state with multiple such objects in a sequence over the course of a single time step.

1.4 Frequently Asked Questions

1.4.1 Isn't Python too slow for Earth System Models?

Not in general. Most model run time is spent within code such as the dynamical core, radiation parameterization, and other physics parameterizations. These components can be written in your favorite compiled language like Fortran or C, and then run from within Python. For new projects where you're writing a component from scratch, we recommend **Cython**, as it allows you to write typed Python code which gets converted into C code and then compiled. Sympl is designed so that only overhead tasks need to be written in Python.

If 90% of a model's run time is spent within this computationally intensive, compiled code, and the other 10% is spent in overhead code, then that overhead code taking 3x as long to run would only increase the model's run time by 1/5th.

But the run time of a model isn't the only important aspect, you also have to consider time spent programming a model. Poorly designed and documented code can cost weeks of researcher time. It can also take a long time to perform tasks that Sympl makes easy, like porting a component from one model to another. Time is also saved when others have to read and understand your model code.

In short, the more your work involves configuring and developing models, the more time you will save, at the cost of slightly slower model runs. But in the end, what is the cost of your sanity?

1.4.2 What calendar is my model using?

Hopefully the section on *Choice of Datetime* can clear this up.

1.5 Installation

1.5.1 Latest release

To install Sympl, run this command in your terminal:

```
$ pip install sympl
```

This is the preferred method to install Sympl, as it will always install the most recent release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.5.2 From sources

The sources for Sympl can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/mcgibbon/sympl
```

Or download the tarball:

```
$ curl  -OL https://github.com/mcgibbon/sympl/tarball/master
```

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

```
$ python setup.py install
```

If you are looking to modify the code, you can install it with:

```
$ python setup.py develop
```

This configures the package so that Python points to the current directory instead of copying files. Then when you make modifications to the source code in that directory, they are automatically used by any new Python sessions.

1.6 Model State

In a Sympl model, physical quantities are stored in a state dictionary. This is a Python **dict** with keys that are strings, indicating the quantity name, and values are *DataArray* objects. The *DataArray* is a slight modification of the

`DataArray` object from xarray. It maintains attributes when it is on the left hand side of addition or subtraction, and contains a helpful method for converting units. Any information about the grid the data is using that components need should be put as attributes in the `attrs` of the `DataArray` objects. Deciding on these attributes (if any) is mostly up to the component developers. However, in order to use the TimeStepper objects and several helper functions from Sympl, it is required that a "units" attribute is present.

class sympl.**DataArray**(*data*, *coords=None*, *dims=None*, *name=None*, *attrs=None*, *encoding=None*, *fastpath=False*)

__add__(*other*)
If this DataArray is on the left side of the addition, keep its attributes when adding to the other object.

__sub__(*other*)
If this DataArray is on the left side of the subtraction, keep its attributes when subtracting the other object.

to_units(*units*)
Convert the units of this DataArray, if necessary. No conversion is performed if the units are the same as the units of this DataArray. The units of this DataArray are determined from the "units" attribute in attrs.

Parameters **units** (*str*) – The desired units.

Raises

- `ValueError` – If the units are invalid for this object.
- `KeyError` – If this object does not have units information in its attrs.

Returns **converted_data** – A DataArray containing the data from this object in the desired units, if possible.

Return type *DataArray*

There is one quantity which is not stored as a *DataArray*, and that is "time". Time must be stored as a datetime or timedelta-like object.

Code to initialize the state is intentionally not present in Sympl, since this depends heavily on the details of the model you are running. You may find helper functions to create an initial state in model packages, or you can write your own. For example, below you can see code to initialize a state with random temperature and pressure on a lat-lon grid (random values are used for demonstration purposes only, and are not recommended in a real model).

```
from datetime import datetime
import numpy as np
from sympl import DataArray, add_direction_names
n_lat = 64
n_lon = 128
n_height = 32
add_direction_names(x='lat', y='lon', z=('mid_levels', 'interface_levels'))
state = {
    "time": datetime(2000, 1, 1),
    "air_temperature": DataArray(
        np.random.rand(n_lat, n_lon, n_height),
        dims=('lat', 'lon', 'mid_levels'),
        attrs={'units': 'degK'}),
    "air_pressure": DataArray(
        np.random.rand(n_lat, n_lon, n_height),
        dims=('lat', 'lon', 'mid_levels'),
        attrs={'units': 'Pa'}),
    "air_pressure_on_interface_levels": DataArray(
        np.random.rand(n_lat, n_lon, n_height + 1),
        dims=('lat', 'lon', 'interface_levels'),
```

(continues on next page)

(continued from previous page)

```
        attrs={'units': 'Pa'}),
    }
```

The call to `add_direction_names()` tells Sympl what dimension names correspond to what directions. This information is used by components to make sure the axes are in the right order.

1.6.1 Choice of Datetime

The built-in `datetime` object in Python (as used above) assumes the proleptic Gregorian calendar, which extends the Gregorian calendar back infinitely. Sympl provides a *`datetime()`* function which returns a datetime-like object, and allows a variety of different calendars. If a calendar other than 'proleptic_gregorian' is specified, one of the classes from the netcdftime package will be used. Of course, this requires that it is installed! If it's not, you will get an error, and should `pip install netcdftime`. Sympl also includes *`timedelta`* for convenience. This is just the default Python `timedelta`.

To repeat, the calendar your model is using depends entirely on what object you're using to store time in the state dictionary, and the default one uses the proleptic Gregorian calendar used by the default Python `datetime`.

`sympl.`**`datetime`**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *calendar='proleptic_gregorian'*)

Retrieves a datetime-like object with the requested calendar. Calendar types other than proleptic_gregorian require the netcdftime module to be installed.

Parameters

- **`year`** (*`int,`*) –
- **`month`** (*`int,`*) –
- **`day`** (*`int,`*) –
- **`hour`** (*`int, optional`*) –
- **`minute`** (*`int, optional`*) –
- **`second`** (*`int, optional`*) –
- **`microsecond`** (*`int, optional`*) –
- **`tzinfo`** (*`datetime.tzinfo, optional`*) – A timezone informaton class, such as from pytz. Can only be used with 'proleptic_gregorian' calendar, as netcdftime does not support timezones.
- **`calendar`** (*`string, optional`*) – Should be one of 'proleptic_gregorian', 'no_leap', '365_day', 'all_leap', '366_day', '360_day', 'julian', or 'gregorian'. Default is 'proleptic_gregorian', which returns a normal Python datetime. Other options require the netcdftime module to be installed.

Returns **datetime** – The requested datetime. May be a Python datetime, or one of the datetime-like types in netcdftime.

Return type datetime-like

`class` `sympl.`**`timedelta`**

Difference between two datetime values.

1.6.2 Naming Quantities

If you are a model user, the names of your quantities should coincide with the names used by the components you are using in your model. Basically, the components you are using dictate what quantity names you must use. If you are a model developer, we have a set of guidelines for naming quantities.

Note: The following is intended for model developers.

All quantity names should be verbose, and fully descriptive. Within a component you can set a quantity to an abbreviated variable, such as

```
theta = state['air_potential_temperature']
```

This ensures that your code is self-documenting. It is immediately apparent to anyone reading your code that theta refers to potential temperature of air, even if they are not familiar with theta as a common abbreviation.

We strongly recommend using the standard names according to CF conventions. In addition to making sure your code is self-documenting, this helps make sure that different components are compatible with one another, since they all need to use the same name for a given quantity in the model state.

If your quantity is on vertical interface levels, you should name it using the form "<name>_on_interface_levels". If this is not specified, it is assumed that the quantity is on vertical mid levels. This is necessary because the same quantity may be specified on both mid and interface levels in the same model state.

When in doubt about names, look at what other components have been written that use the same quantity. If it looks like their name is verbose and follows the CF conventions then you should probably use the same name.

1.7 Constants

Configuration is an important part of any modelling framework. In Sympl, component-specific configuration is given to components directly. However, configuration values that may be shared by more than one component are stored as constants. Good examples of these are physical constants, such as gravitational_acceleration, or constants specifying processor counts.

1.7.1 Getting and Setting Constants

You can retrieve and set constants using *get_constant()* and *set_constant()*. *set_constant()* will allow you to set constants regardless of whether a value is already defined for that constant, allowing you to add new constants we haven't thought of.

The constant library can be reverted to its original state when Sympl is imported by calling *reset_constants()*.

sympl.**get_constant**(*name*, *units*)
Retrieves the value of a constant.

Parameters

- **name** (*str*) – The name of the constant.
- **units** (*str*) – The units requested for the returned value.

Returns **value** – The value of the constant in the requested units.

Return type float

sympl.**set_constant**(*name*, *value*, *units*)
Sets the value of a constant.

Parameters

- **name** (*str*) – The name of the constant.
- **value** (*float*) – The value to which the constant should be set.
- **units** (*str*) – The units of the value given.

sympl.**reset_constants**()
Reverts constants to their state when Sympl was originally imported. This includes removing any new constants, setting the original constants to their original values, and setting the condensible quantity to water.

1.7.2 Condensible Quantities

For Earth system modeling, water is used as a condensible compound. By default, condensible quantities such as 'density_of_ice' and 'heat_capacity_of_liquid_phase' are aliases for the corresponding value for water. If you would like to use a different condensible compound, you can use the *set_condensible_name()* function. For example:

```
import sympl
sympl.set_condensible_name('carbon_dioxide')
sympl.get_constant('heat_capacity_of_solid_phase', 'J kg^-1 K^-1')
```

will set the condensible compound to carbon dioxide, and then get the heat capacity of solid carbon dioxide (if it has been set). For example, the constant name 'heat_capacity_of_solid_phase' would then be an alias for 'heat_capacity_of_solid_carbon_dioxide'.

When setting the value of an alias, the value of the aliased quantity is the one which will be altered. For example, if you run

```
import sympl
sympl.set_constant('heat_capacity_of_liquid_phase', 1.0, 'J kg^-1 K^-1')
```

you would change the heat capacity of liquid water (since water is the default condensible compound).

sympl.**set_condensible_name**(*name*)

1.7.3 Default Constants

The following constants are available in Sympl by default:

class sympl._core.constants.**ConstantList**

Condensible density_of_liquid_phase: 1000.0 kg m^-3

heat_capacity_of_liquid_phase: 4185.5 J kg^-1 K^-1

heat_capacity_of_vapor_phase: 1846.0 J kg^-1 K^-1

specific_enthalpy_of_vapor_phase: 2500.0 J kg^-1

gas_constant_of_vapor_phase: 461.5 J kg^-1 K^-1

latent_heat_of_condensation: 2500000.0 J kg^-1

latent_heat_of_fusion: 333550.0 J kg^-1

density_of_solid_phase_as_ice: 916.7 kg m^-3

density_of_solid_phase_as_snow: 100.0 kg m^-3

heat_capacity_of_solid_phase_as_ice: 2108.0 J kg^-1 K^-1

heat_capacity_of_solid_phase_as_snow: 2108.0 J kg^-1 K^-1

thermal_conductivity_of_solid_phase_as_ice: 2.22 W m^-1 K^-1

thermal_conductivity_of_solid_phase_as_snow: 0.2 W m^-1 K^-1

thermal_conductivity_of_liquid_phase: 0.57 W m^-1 K^-1

freezing_temperature_of_liquid_phase: 273.0 K

Stellar stellar_irradiance: 1367.0 W m^-2

Atmospheric heat_capacity_of_dry_air_at_constant_pressure: 1004.64 J kg^-1 K^-1

gas_constant_of_dry_air: 287.0 J kg^-1 K^-1

thermal_conductivity_of_dry_air: 0.026 W m^-1 K^-1

reference_air_pressure: 101320.0 Pa

Planetary gravitational_acceleration: 9.80665 m s^-2

planetary_radius: 6371000.0 m

planetary_rotation_rate: 7.292e-05 s^-1

seconds_per_day: 86400.0

Chemical heat_capacity_of_water_vapor_at_constant_pressure: 1846.0 J kg^-1 K^-1

density_of_liquid_water: 1000.0 kg m^-3

gas_constant_of_water_vapor: 461.5 J kg^-1 K^-1

latent_heat_of_vaporization_of_water: 2500000.0 J kg^-1

heat_capacity_of_liquid_water: 4185.5 J kg^-1 K^-1

latent_heat_of_fusion_of_water: 333550.0 J kg^-1

Physical stefan_boltzmann_constant: 5.670367e-08 W m^-2 K^-4

avogadro_constant: 6.022140857e+23 mole^-1

speed_of_light: 299792458.0 m s^-1

boltzmann_constant: 1.38064852e-23 J K^-1

loschmidt_constant: 2.6516467e+25 m^-3

universal_gas_constant: 8.3144598 J mole^-1 K^-1

planck_constant: 6.62607004e-34 J s

1.8 Timestepping

TimeStepper objects use time derivatives from *Prognostic* objects to step a model state forward in time. They are initialized using a list of *Prognostic* objects.

```
from sympl import AdamsBashforth
time_stepper = AdamsBashforth([MyPrognostic(), MyOtherPrognostic()])
```

Once initialized, a *TimeStepper* object has a very similar interface to the *Implicit* object.

```
from datetime import timedelta
time_stepper = AdamsBashforth([MyPrognostic()])
timestep = timedelta(minutes=10)
diagnostics, next_state = time_stepper(state, timestep)
state.update(diagnostics)
```

The returned `diagnostics` dictionary contains diagnostic quantities from the timestep of the input `state`, while `next_state` is the state dictionary for the next timestep. It is possible that some of the arrays in `diagnostics` may be the same arrays as were given in the input `state`, and that they have been modified. In other words, `state` may be modified by this call. For instance, the time filtering necessary when using Leapfrog time stepping means the current model state has to be modified by the filter.

It is only after calling the *TimeStepper* and getting the diagnostics that you will have a complete state with all diagnostic quantities. This means you will sometimes want to pass `state` to your *Monitor* objects *after* calling the *TimeStepper* and getting `next_state`.

Warning: *TimeStepper* objects do not, and should not, update 'time' in the model state.

Keep in mind that for split-time models, multiple *TimeStepper* objects might be called in in a single pass of the main loop. If each one updated `state['time']`, the time would be moved forward more than it should. For that reason, *TimeStepper* objects do not update `state['time']`.

There are also *Implicit* objects which evolve the state forward in time without the use of Prognostic objects. These function exactly the same as a *TimeStepper* once they are created, but do not accept *Prognostic* objects when you create them. One example might be a component that condenses all supersaturated moisture over some time period. *Implicit* objects are generally used for parameterizations that work by determining the target model state in some way, or involve limiters, and cannot be represented as a *Prognostic*.

class sympl.**TimeStepper**(*prognostic_list*, ***kwargs*)

An object which integrates model state forward in time.

It uses Prognostic and Diagnostic objects to update the current model state with diagnostics, and to return the model state at the next timestep.

inputs
tuple of str – The quantities required in the state when the object is called.

diagnostics
tuple of str – The quantities for which values for the old state are returned when the object is called.

outputs
tuple of str – The quantities for which values for the new state are returned when the object is called.

__call__(*state*, *timestep*)

Retrieves any diagnostics and returns a new state corresponding to the next timestep.

Parameters

- **state** (*dict*) – The current model state.
- **timestep** (timedelta) – The amount of time to step forward.

Returns

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – The model state at the next timestep.

__init__(*prognostic_list*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

class sympl.**AdamsBashforth**(*prognostic_list*, *order=3*)
A TimeStepper using the Adams-Bashforth scheme.

__call__(*state*, *timestep*)
Updates the input state dictionary and returns a new state corresponding to the next timestep.

Parameters

- **state** (*dict*) – The current model state. Will be updated in-place by the call with any diagnostics from the current timestep.
- **timestep** (timedelta) – The amount of time to step forward.

Returns

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – The model state at the next timestep.

Raises ValueError – If the timestep is not the same as the last time step() was called on this instance of this object.

__init__(*prognostic_list*, *order=3*)
Initialize an Adams-Bashforth time stepper.

Parameters

- **prognostic_list** (*iterable of Prognostic*) – Objects used to get tendencies for time stepping.
- **order** (*int, optional*) – The order of accuracy to use. Must be between 1 and 4. 1 is the same as the Euler method. Default is 3.

class sympl.**Leapfrog**(*prognostic_list*, *asselin_strength=0.05*, *alpha=0.5*)
A TimeStepper using the Leapfrog scheme.

This scheme calculates the values at time t_{n+1} using the derivatives at t_{n} and values at t_{n-1}. Following the step, an Asselin filter is applied to damp the computational mode that results from the scheme and maintain stability. The Asselin filter brings the values at t_{n} (and optionally the values at t_{n+1}, according to Williams (2009)) closer to the mean of the values at t_{n-1} and t_{n+1}.

__call__(*state*, *timestep*)
Updates the input state dictionary and returns a new state corresponding to the next timestep.

Parameters

- **state** (*dict*) – The current model state. Will be updated in-place by the call due to the Robert-Asselin-Williams filter.
- **timestep** (timedelta) – The amount of time to step forward.

Returns

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – The model state at the next timestep.

Raises

- `SharedKeyError` – If a Diagnostic object has an output that is already in the state at the start of the timestep.
- `ValueError` – If the timestep is not the same as the last time step() was called on this instance of this object.

__init__ (*prognostic_list*, *asselin_strength=0.05*, *alpha=0.5*)
Initialize a Leapfrog time stepper.

Parameters

- **prognostic_list** (*iterable of Prognostic*) – Objects used to get tendencies for time stepping.
- **asselin_strength** (*float, optional*) – The filter parameter used to determine the strength of the Asselin filter. Default is 0.05.
- **alpha** (*float, optional*) – Constant from Williams (2009), where the midpoint is shifted by alpha*influence, and the right point is shifted by (1-alpha)*influence. If alpha is 1 then the behavior is that of the classic Robert-Asselin time filter, while if it is 0.5 the filter will conserve the three-point mean. Default is 0.5.

References

Williams, P., 2009: A Proposed Modification to the Robert-Asselin Time Filter. Mon. Wea. Rev., 137, 2538–2546, doi: 10.1175/2009MWR2724.1.

1.9 Component Types

In Sympl, computation is mainly performed using *Prognostic*, *Diagnostic*, and *Implicit* objects. Each of these types, once initialized, can be passed in a current model state. *Prognostic* objects use the state to return tendencies and diagnostics at the current time. *Diagnostic* objects return only diagnostics from the current time. *Implicit* objects will take in a timestep along with the state, and then return the next state as well as modifying the current state to include more diagnostics (it is similar to a *TimeStepper* in how it is called).

In specific cases, it may be necessary to use a *ImplicitPrognostic* object, which is discussed at the end of this section.

These classes themselves (listed in the previous paragraph) are not ones you can initialize (e.g. there is no one 'prognostic' scheme), but instead should be subclassed to contain computational code relevant to the model you're running.

In addition to the computational functionality below, all components have "properties" for their inputs and outputs, which are described in the section *Input/Output Properties*.

1.9.1 Prognostic

As stated above, *Prognostic* objects use the state to return tendencies and diagnostics at the current time. In a full model, the tendencies are used by a time stepping scheme (in Sympl, a *TimeStepper*) to determine the values of quantities at the next time.

You can call a *Prognostic* directly to get diagnostics and tendencies like so:

```
radiation = RRTMRadiation()
diagnostics, tendencies = radiation(state)
```

`diagnostics` and `tendencies` in this case will both be dictionaries, similar to `state`. Even if the *`Prognostic`* being called does not compute any diagnostics, it will still return an empty diagnostics dictionary.

Usually, you will call a Prognostic object through a *`TimeStepper`* that uses it to determine values at the next timestep.

class sympl.**Prognostic**

inputs
tuple of str – The quantities required in the state when the object is called.

tendencies
tuple of str – The quantities for which tendencies are returned when the object is called.

diagnostics
tuple of str – The diagnostic quantities returned when the object is called.

input_properties
dict – A dictionary whose keys are quantities required in the state when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

tendency_properties
dict – A dictionary whose keys are quantities for which tendencies are returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

diagnostic_properties
dict – A dictionary whose keys are diagnostic quantities returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

__call__(*state*)
Gets tendencies and diagnostics from the passed model state.

Parameters **state** (*dict*) – A model state dictionary.

Returns

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

Raises

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Prognostic instance.

__repr__()
Return repr(self).

__str__()
Return str(self).

class sympl.**ConstantPrognostic**(*tendencies*, *diagnostics=None*)
Prescribes constant tendencies provided at initialization.

Note: Any arrays in the passed dictionaries are not copied, so that if you were to modify them after passing them into this object, it would also modify the values inside this object.

__call__(*state*)
Gets tendencies and diagnostics from the passed model state. The returned dictionaries will contain the same values as were passed at initialization.

Parameters **state** (*dict*) – A model state dictionary.

Returns

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities.

__init__(*tendencies*, *diagnostics=None*)

Parameters

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second to be returned by this Prognostic.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities to be returned by this Prognostic.

class sympl.**RelaxationPrognostic**(*quantity_name*, *equilibrium_value=None*, *relaxation_timescale=None*)
Applies Newtonian relaxation to a single quantity.

The relaxation takes the form $\frac{dx}{dt} = -\frac{x - x_{eq}}{\tau}$ where x is the quantity being relaxed, x_{eq} is the equilibrium value, and τ is the timescale of the relaxation.

__call__(*state*)
Gets tendencies and diagnostics from the passed model state.

Parameters **state** (*dict*) – A model state dictionary. Below, (quantity_name) refers to the quantity_name passed at initialization. The state must contain:

- (quantity_name)
- equilibrium_(quantity_name), unless this was passed at initialisation time in which case that value is used
- (quantity_name)_relaxation_timescale, unless this was passed at initialisation time in which case that value is used

Returns

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

__init__(*quantity_name*, *equilibrium_value=None*, *relaxation_timescale=None*)

Parameters

- **quantity_name** (*str*) – The name of the quantity to which Newtonian relaxation should be applied
- **equilibrium_value** (DataArray, *optional*) – The equilibrium value to which the quantity is relaxed. If not given, it should be provided in the state when the object is called.

- **relaxation_timescale** (*DataArray, optional*) – The timescale tau with which the Newtonian relaxation occurs. If not given, it should be provided in the state when the object is called.

1.9.2 Diagnostic

Diagnostic objects use the state to return quantities ('diagnostics') from the same timestep as the input state. You can call a *Diagnostic* directly to get diagnostic quantities like so:

```
diagnostic_component = MyDiagnostic()
diagnostics = diagnostic_component(state)
```

You should be careful to check in the documentation of the particular *Diagnostic* you are using to see whether it modifies the `state` given to it as input. *Diagnostic* objects in charge of updating ghost cells in particular may modify the arrays in the input dictionary, so that the arrays in the returned `diagnostics` dictionary are the same ones as were sent as input in the `state`. To make it clear that the state is being modified when using such objects, we recommend using a syntax like:

```
state.update(diagnostic_component(state))
```

class sympl.**Diagnostic**

inputs
tuple of str – The quantities required in the state when the object is called.

diagnostics
tuple of str – The diagnostic quantities returned when the object is called.

input_properties
dict – A dictionary whose keys are quantities required in the state when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

diagnostic_properties
dict – A dictionary whose keys are diagnostic quantities returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

__call__(*state*)
Gets diagnostics from the passed model state.

Parameters **state** (*dict*) – A model state dictionary.

Returns **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

Return type dict

Raises

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Prognostic instance.

__repr__()
Return repr(self).

__str__()
Return str(self).

class sympl.**ConstantDiagnostic**(*diagnostics*)
Yields constant diagnostics provided at initialization.

Note: Any arrays in the passed dictionaries are not copied, so that if you were to modify them after passing them into this object, it would also modify the values inside this object.

__call__(*state*)
Returns diagnostic values.

Parameters **state** (*dict*) – A model state dictionary. Is not used, and is only taken in to keep an API consistent with a Diagnostic.

Returns **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities. The values in the returned dictionary are the same as were passed into this object at initialization.

Return type dict

__init__(*diagnostics*)

Parameters **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities. The values in the dictionary will be returned when this Diagnostic is called.

1.9.3 Implicit

Implicit objects use a state and a timestep to return the next state, and update the input state with any relevant diagnostic quantities. You can call an Implicit object like so:

```
from datetime import timedelta
implicit = MyImplicit()
timestep = timedelta(minutes=10)
diagnostics, next_state = implicit(state, timestep)
state.update(diagnostics)
```

The returned `diagnostics` dictionary contains diagnostic quantities from the timestep of the input `state`, while `next_state` is the state dictionary for the next timestep. It is possible that some of the arrays in `diagnostics` may be the same arrays as were given in the input `state`, and that they have been modified. In other words, `state` may be modified by this call. For instance, the object may need to update ghost cells in the current state. Or if an object provides 'cloud_fraction' as a diagnostic, it may modify an existing 'cloud_fraction' array in the input state if one is present, instead of allocating a new array.

class sympl.**Implicit**

inputs
tuple of str – The quantities required in the state when the object is called.

diagnostics
tuple of str – The quantities for which values for the old state are returned when the object is called.

outputs
tuple of str – The quantities for which values for the new state are returned when the object is called.

input_properties
dict – A dictionary whose keys are quantities required in the state when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

diagnostic_properties
dict – A dictionary whose keys are quantities for which values for the old state are returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

output_properties
dict – A dictionary whose keys are quantities for which values for the new state are returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

__call__(*state*, *timestep*)
Gets diagnostics from the current model state and steps the state forward in time according to the timestep.

Parameters

- **state** (*dict*) – A model state dictionary. Will be updated with any diagnostic quantities produced by this object for the time of the input state.
- **timestep** (timedelta) – The amount of time to step forward.

Returns

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the timestep after input state.

Raises

- KeyError – If a required quantity is missing from the state.
- InvalidStateError – If state is not a valid input for the Implicit instance for other reasons.

__repr__()
Return repr(self).

__str__()
Return str(self).

1.9.4 Input/Output Properties

You may have noticed when reading the documentation for the classes above that there are a number of attributes with names like input_properties for components. These attributes give a fairly complete description of the inputs and outputs of the component.

You can access them like this (for an example *Prognostic* class RRTMRadiation):

```
radiation = RRTMRadiation()
radiation.input_properties
radiation.diagnostic_properties
radiation.tendency_properties
```

Input

All components have input_properties, because they all take inputs. This attribute (like all the other properties attributes) is a python dict, or "dictionary" (if you are unfamiliar with these, please read the Python documentation for dicts).

An example input_properties would be

```
{
    'air_temperature': {
        'dims': ['*', 'z'],
        'units': 'degK',
    },
    'vertical_wind': {
        'dims': ['*', 'z'],
        'units': 'm/s',
        'match_dims_like': ['air_temperature']
    }
}
```

Each entry in the input_properties dictionary is a quantity that the object requires as an input, and its value is another dictionary that tells you how the object uses that quantity. The `units` property is the units used internally in the object. You don't need to pass in the quantity with those those units, as long as the units can be converted, but if you do use the same units in the input state it will avoid the computational cost of converting units.

The `dims` property can be more confusing, but is very useful. It says what dimensions the component uses internally for those quantities. The component requires that you give it quantities that can be transformed into those internal dimensions, but it can take care of that transformation itself. In this example, it will transform the arrays for both quantities to put the vertical dimension last, and collect all the other dimensions into a single first dimension. If you pass this object arrays that have their vertical dimension last, it may speed up the computation, depending on the component (but not for all components!).

So what are '*' and 'z' anyways? These are *wildcard* dimensions. 'z' will match any dimension that is vertical, while '*' will match *any* dimension that is not specified somewhere else in the `dims` list. There are also 'x' and 'y' for horizontal dimensions. The directional matches are given to Sympl using the functions `set_direction_names()` or `add_direction_names()`. If you're using someone else's package for a component, it is likely that they call these functions for you, so you don't have to (and if you're writing such a package, you should use `add_direction_names()`).

If a component is using a wildcard it means it doesn't care very much about those directions. For example, a column component like radiation will simply call itself on each column of the domain, so it doesn't care about the specifics of what the non-vertical dimensions are, as long as the desired quantities are co-located.

That's where `match_dims_like` comes in. This property says the object requires all shared wildcard dimensions between the two quantity match the same dimensions as the other specified quantity. In this case, it will ensure that `vertical_wind` is on the same grid as `air_temperature`.

Let's consider a slight variation on the earlier example:

```
{
    'air_temperature': {
        'dims': ['*', 'mid_levels'],
        'units': 'degK',
    },
    'vertical_wind': {
        'dims': ['*', 'interface_levels'],
        'units': 'm/s',
        'match_dims_like': ['air_temperature']
    }
}
```

This version requires that `air_temperature` be on the `mid_levels` vertical grid, while `vertical_wind` is on the `interface_levels`. It still requires that all other dimensions are the same between the two quantities, so that they are on the same horizontal grid (if they have a horizontal grid).

Outputs

There are a few output property dictionaries in Sympl: `tendency_properties`, `diagnostic_properties`, and `output_properties`. They are all formatted the same way with the same properties, but tell you about the tendencies, diagnostics, or next state values that are output by the component, respectively.

Here's an example output dictionary:

```
tendency_properties = {
    'air_temperature': {
        'dims_like': 'air_temperature',
        'units': 'degK/s',
    }
}
```

In tendency_properties, the quantity names specify the quantities for which tendencies are given. The `units` are the units of the output value, which is also put in the output *DataArray* as the `units` attribute.

`dims_like` is telling you that the output array will have the same dimensions as the array you gave it for `air_temperature` as an input. If you pass it an `air_temperature` array with ('latitude', 'longitude', 'mid_levels') as its axes, it will return an array with ('latitude', 'longitude', 'mid_levels') for the temperature tendency. If `dims_like` is not specified in the `tendency_properties` dictionary, it is assumed to be the matching quantity in the input, but for the other quantities `dims_like` must always be explicitly defined. For instance, if the object as a `diagnostic_properties` equal to:

```
diagnostic_properties = {
    'cloud_fraction': {
        'dims_like': 'air_temperature',
        'units': '',
    }
}
```

that the object will output `cloud_fraction` in its diagnostics on the same grid as `air_temperature`, in dimensionless units.

1.9.5 ImplicitPrognostic

Warning: This component type should be avoided unless you know you need it, for reasons discussed in this section.

In addition to the component types described above, computation may be performed by a *ImplicitPrognostic*. This class should be avoided unless you know what you are doing, but it may be necessary in certain cases. An *ImplicitPrognostic*, like a *Prognostic*, calculates tendencies, but it does so using both the model state and a timestep. Certain components, like ones handling advection using a spectral method, may need to derive tendencies from an *Implicit* object by representing it using an *ImplicitPrognostic*.

The reason to avoid using an *ImplicitPrognostic* is that if a component requires a timestep, it is making internal assumptions about how you are timestepping. For example, it may use the timestep to ensure that all supersaturated water is condensed by the end of the timestep using an assumption about the timestepping. However, if you use a *TimeStepper* which does not obey those assumptions, you may get unintended behavior, such as some supersaturated water remaining, or too much water being condensed.

For this reason, the *TimeStepper* objects included in Sympl do not wrap *ImplicitPrognostic* components. If you would like to use this type of component, and know what you are doing, it is pretty easy to write your own

TimeStepper to do so (you can base the code off of the code in Sympl), or the model you are using might already have components to do this for you.

If you are wrapping a parameterization and notice that it needs a timestep to compute its tendencies, that is likely *not* a good reason to write an *ImplicitPrognostic*. If at all possible you should modify the code to compute the value at the next timestep, and write an *Implicit* component. You are welcome to reach out to the developers of Sympl if you would like advice on your specific situation! We're always excited about new wrapped components.

class sympl.**ImplicitPrognostic**

inputs
tuple of str – The quantities required in the state when the object is called.

tendencies
tuple of str – The quantities for which tendencies are returned when the object is called.

diagnostics
tuple of str – The diagnostic quantities returned when the object is called.

input_properties
dict – A dictionary whose keys are quantities required in the state when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

tendency_properties
dict – A dictionary whose keys are quantities for which tendencies are returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

diagnostic_properties
dict – A dictionary whose keys are diagnostic quantities returned when the object is called, and values are dictionaries which indicate 'dims' and 'units'.

__call__(*state*, *timestep*)
Gets tendencies and diagnostics from the passed model state.

Parameters

- **state** (*dict*) – A model state dictionary.
- **timestep** (timedelta) – The time over which the model is being stepped.

Returns

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

Raises

- KeyError – If a required quantity is missing from the state.
- InvalidStateError – If state is not a valid input for the Prognostic instance.

__repr__()
Return repr(self).

__str__()
Return str(self).

1.10 Monitors

Monitor objects store states in some way, whether it is by displaying the new state on a plot that is shown to the user, updating information on a web server, or saving the state to a file. They are called like so:

```
monitor = MyMonitor()
monitor.store(state)
```

The *Monitor* will take advantage of the 'time' key in the `state` dictionary in order to determine the model time of the state. This is particularly important for a *Monitor* which outputs a series of states to disk.

class sympl.**Monitor**

__repr__()
Return repr(self).

__str__()
Return str(self).

store(*state*)
Stores the given state in the Monitor and performs class-specific actions.

Parameters **state** (*dict*) – A model state dictionary.

Raises `InvalidStateError` – If state is not a valid input for the Diagnostic instance.

class sympl.**NetCDFMonitor**(*filename*, *time_units='seconds'*, *store_names=None*, *write_on_store=False*, *aliases=None*)
A Monitor which caches stored states and then writes them to a NetCDF file when requested.

__init__(*filename*, *time_units='seconds'*, *store_names=None*, *write_on_store=False*, *aliases=None*)

Parameters

- **filename** (*str*) – The file to which the NetCDF file will be written.
- **time_units** (*str, optional*) – The units in which time will be stored in the NetCDF file. Time is stored as an integer number of these units. Default is seconds.
- **store_names** (*iterable of str, optional*) – Names of quantities to store. If not given, all quantities are stored.
- **write_on_store** (*bool, optional*) – If True, stored changes are immediately written to file. This can result in many file open/close operations. Default is to write only when the write() method is called directly.
- **aliases** (*dict*) – A dictionary of string replacements to apply to state variable names before saving them in netCDF files.

store(*state*)
Caches the given state. If write_on_store=True was passed on initialization, also writes to file. Normally a call to the write() method is required to write to file.

Parameters **state** (*dict*) – A model state dictionary.

Raises `InvalidStateError` – If state is not a valid input for the Diagnostic instance.

write()
Write all cached states to the NetCDF file, and clear the cache. This will append to any existing NetCDF file.

Raises `InvalidStateError` – If cached states do not all have the same quantities as every other cached and written state.

class sympl.**PlotFunctionMonitor**(*plot_function*, *interactive=True*)

A Monitor which uses a user-defined function to draw figures using model state.

__init__(*plot_function*, *interactive=True*)

Initialize a PlotFunctionMonitor.

Parameters

- **plot_function** (*func*) – A function plot_function(fig, state) that draws the given state onto the given (initially clear) figure.
- **interactive** (*bool, optional*) – If true, matplotlib's interactive mode will be enabled, allowing plot animation while other computation is running.

store(*state*)

Updates the plot using the given state.

Parameters **state** (*dict*) – A model state dictionary.

1.11 Composites

There are a set of objects in Sympl that wrap multiple components into a single object so they can be called as if they were one component. There is one each for *Prognostic*, *Diagnostic*, and *Monitor*. These can be used to simplify code, so that the way you call a list of components is the same as the way you would call a single component. For example, *instead* of writing:

```
prognostic_list = [
    MyPrognostic(),
    MyOtherPrognostic(),
    YetAnotherPrognostic(),
]
all_diagnostics = {}
total_tendencies = {}
for prognostic_component in prognostic_list:
    tendencies, diagnostics = prognostic_component(state)
    # this should actually check to make sure nothing is overwritten,
    # but this code does not
    total_tendencies.update(tendencies)
    for name, value in tendencies.keys():
        if name not in total_tendencies:
            total_tendencies[name] = value
        else:
            total_tendencies[name] += value
    for name, value in diagnostics.items():
        all_diagnostics[name] = value
```

You could write:

```
prognostic_composite = PrognosticComposite([
    MyPrognostic(),
    MyOtherPrognostic(),
    YetAnotherPrognostic(),
])
tendencies, diagnostics = prognostic_composite(state)
```

This second call is much cleaner. It will also automatically detect whether multiple components are trying to write out the same diagnostic, and raise an exception if that is the case (so no results are being silently overwritten). You can get similar simplifications for *Diagnostic* and *Monitor*.

Note: PrognosticComposites are mainly useful inside of TimeSteppers, so if you're only writing a model script it's unlikely you'll need them.

1.11.1 API Reference

class sympl.**PrognosticComposite**(**args*)

inputs
tuple of str – The quantities required in the state when the object is called.

tendencies
tuple of str – The quantities for which tendencies are returned when the object is called.

diagnostics
tuple of str – The diagnostic quantities returned when the object is called.

__call__(*state*)
Gets tendencies and diagnostics from the passed model state.

Parameters **state** (*dict*) – A model state dictionary.

Returns

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

Raises

- SharedKeyError – If multiple Prognostic objects contained in the collection return the same diagnostic quantity.
- KeyError – If a required quantity is missing from the state.
- InvalidStateError – If state is not a valid input for a Prognostic instance.

component_class
alias of *Prognostic*

class sympl.**DiagnosticComposite**(**args*)

inputs
tuple of str – The quantities required in the state when the object is called.

diagnostics
tuple of str – The diagnostic quantities returned when the object is called.

__call__(*state*)
Gets diagnostics from the passed model state.

Parameters **state** (*dict*) – A model state dictionary.

Returns **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

Return type dict

Raises

- `SharedKeyError` – If multiple Diagnostic objects contained in the collection return the same diagnostic quantity.
- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for a Diagnostic instance.

component_class
alias of *Diagnostic*

class sympl.**MonitorComposite**(**args*)

store(*state*)
Stores the given state in the Monitor and performs class-specific actions.

Parameters **state** (*dict*) – A model state dictionary.

Raises

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for a Monitor instance.

1.12 Writing Components

Note: This section is intended for model developers. If you intend to use only components that are already written, you can probably ignore it.

Perhaps the best way to learn how to write components is to read components someone else has written. For example, you can look at the CliMT project. Here we will go over a couple examples of physically simple, made-up components to talk about the parts of their code.

1.12.1 Writing an Example

Let's start with a Prognostic component which relaxes temperature towards some target temperature.

```
from sympl import (
    Prognostic, get_numpy_arrays_with_properties,
    restore_data_arrays_with_properties)

class TemperatureRelaxation(Prognostic):

    input_properties = {
        'air_temperature': {
            'dims': ['*'],
            'units': 'degK',
        },
        'vertical_wind': {
```

(continues on next page)

(continued from previous page)

```
            'dims': ['*'],
            'units': 'm/s',
            'match_dims_like': ['air_temperature']
        }
    }

    diagnostic_properties = {}

    tendency_properties = {
        'air_temperature': {
            'dims_like': 'air_temperature',
            'units': 'degK/s',
        }
    }

    def __init__(self, tau=1., target_temperature=300.):
        self._tau = tau
        self._T0 = target_temperature

    def __call__(self, state):
        # we get numpy arrays with specifications from input_properties
        raw_arrays = get_numpy_arrays_with_properties(
            state, self.input_properties)
        T = raw_arrays['air_temperature']
        # here the actual computation happens
        raw_tendencies = {
            'air_temperature': (T - self._T0)/self._tau,
        }
        # now we re-format the data in a way the host model can use
        diagnostics = {}
        tendencies = restore_data_arrays_with_properties(
            raw_tendencies, self.tendency_properties,
            state, self.input_properties)
        return tendencies, diagnostics
```

Imports

There are a lot of parts to that code, so let's go through some of them step-by-step. First we have to import objects and functions from Sympl that we plan to use. The import statement should always go at the top of your file so that it can be found right away by anyone reading your code.

```
from sympl import (
    Prognostic, get_numpy_arrays_with_properties,
    restore_data_arrays_with_properties)
```

Define an Object

Once these are imported, there's this line:

```
class TemperatureRelaxation(Prognostic):
```

This is the syntax for defining an object in Python. `TemperatureRelaxation` will be the name of the new object. The *`Prognostic`* in parentheses is telling Python that `TemperatureRelaxation` is a *subclass* of *`Prognostic`*. This tells Sympl that it can expect your object to behave like a *`Prognostic`*.

Define Attributes

The next few lines define attributes of your object:

```
input_properties = {
    'air_temperature': {
        'dims': ['*'],
        'units': 'degK',
    },
    'eastward_wind': {
        'dims': ['*'],
        'units': 'm/s',
        'match_dims_like': ['air_temperature']
    }
}

diagnostic_properties = {}

tendency_properties = {
    'air_temperature': {
        'dims_like': 'air_temperature',
        'units': 'degK/s',
    }
}
```

Note: 'eastward_wind' wouldn't normally make sense as an input for this object, it's only included so we can talk about *match_dims_like*.

These attributes will be attributes both of the class object you're defining and of any instances of that object. That means you can access them using:

```
TemperatureRelaxation.input_properties
```

or on an instance, as when you do:

```
prognostic = TemperatureRelaxation()
prognostic.input_properties
```

These properties are described in *Component Types*. They are very useful! They clearly document your code. Here we can see that air_temperature will be used as a 1-dimensional flattened array in units of degrees Kelvin. Sympl can also understand these properties, and use them to automatically acquire arrays in the dimensions and units that you need. It can also test thatsome of these properties are accurate. It's your responsibility, though, to make sure that the input units are the units you want to acquire in the numpy array data, and that the output units are the units of the values in the raw output arrays that you want to convert to *DataArray* objects.

It is possible that some of these attributes won't be known until you create the object (they may depend on things passed in on initialization). If that's the case, you can write the `__init__` method (see below) so that it sets any relevant properties like `self.input_properties` to have the correct values.

Initialization Method

Next we see a method being defined for this class, which may seem to have a weird name:

```
def __init__(self, damping_timescale=1., target_temperature=300.):
    """
    damping_timescale is the damping timescale in seconds.
    target_temperature is the temperature that will be relaxed to,
    in degrees Kelvin.
    """
    self._tau = damping_timescale
    self._T0 = target_temperature
```

This is the function that is called when you create an instance of your object. All methods on objects take in a first argument called `self`. You don't see it when you call those methods, it gets added in automatically. `self` is a variable that refers to the object on which the method is being called - it's the object itself! When you store attributes on self, as we see in this code, they stay there. You can access them when the object is called later.

Notice some things about the way variables have been named in this `__init__`. The parameters are fairly verbose names which almost fully describe what they are (apart from the units, which are in the documentation string). This is best because it is entirely clear what these values are when others are using your object. You write code for people, not computers! Compilers write code for computers.

Then we take these inputs and store them as attributes with shorter names. This is also optimal. What these attributes mean is clearly defined in the two lines:

```
self._tau = damping_timescale
self._T0 = target_temperature
```

Obviously `self._tau` is the damping timescale, and `self._T0` is the target temperature for the relaxation. Now you can use these shorter variables in the actual code to keep long lines for equations short, knowing that your variables are well-documented.

The Computation

That brings us to the `__call__` method. This is what's called when you use the object as though it is a function. In Sympl components, this is the method which takes in a state dictionary and returns dictionaries with outputs.

```
def __call__(self, state):
    # we get numpy arrays with specifications from input_properties
    raw_arrays = get_numpy_arrays_with_properties(
        state, self.input_properties)
    T = raw_arrays['air_temperature']
    # here the actual computation happens
    raw_tendencies = {
        'air_temperature': (T - self._T0)/self._tau,
    }
    # now we re-format the data in a way the host model can use
    diagnostics = {}
    tendencies = restore_data_arrays_with_properties(
        raw_tendencies, self.tendency_properties,
        state, self.input_properties)
    return diagnostics, tendencies
```

There are two helper functions used in this code that we strongly recommend using. They take care of the work of making sure you get variables that are in the units your component needs, and have the dimensions your component needs.

`get_numpy_arrays_with_properties()` uses the input_properties dictionary you give it to extract numpy arrays with those properties from the input state. It will convert units to ensure the numbers are in the specified units, and it will reshape the data to give it the shape specified in `dims`. For example, if dims is `['*', 'z']`

then it will give you a 2-dimensional array whose second axis is the vertical, and first axis is a flattening of any other dimensions. If you specify `['*', 'mid_levels']` then the result is similar, but only 'mid_levels' is an acceptable vertical dimension. The `match_dims_like` property on `air_pressure` tells Sympl that any wildcard-matched dimensions (ones that match 'x', 'y', 'z', or '*') should be the same between the two quantities, meaning they're on the same grid for those wildcards. You can still, however, have one be on say 'mid_levels' and another on 'interface_levels' if those dimensions are explicitly listed (instead of listing 'z').

restore_data_arrays_with_properties() does something fairly magical. In this example, it takes the raw_tendencies dictionary and converts the value for 'air_temperature' from a numpy array to a DataArray that has the same dimensions as `air_temperature` had in the input state. That means that you could pass this object a state with whatever dimensions you want, whether it's (x, y, z), or (z, x, y), or (x, y), or (station_number, z), etc. and this component will be able to take in that state, and return a tendency dictionary with the same dimensions (and order) that the model uses! And internally you can work with a simple 1-dimensional array. This is particularly useful for writing pointwise components using `['*']` or column components with `['*', 'z']` or `['z', '*']`.

You can read more about properties in the section *Input/Output Properties*.

`sympl.`**`get_numpy_arrays_with_properties`**(*state*, *property_dictionary*)

Parameters

- **`state`** (*dict*) – A state dictionary.
- **`property_dictionary`** (*dict*) – A dictionary whose keys are quantity names and values are dictionaries with properties for those quantities. The property "dims" must be present, indicating the dimensions that the quantity must have when it is returned as a numpy array. The property "units" must be present, and will be used to check the units on the input state and perform a conversion if necessary. If the optional property "match_dims_like" is present, its value should be a quantity also present in property_dictionary, and it will be ensured that any shared wildcard dimensions ('x', 'y', 'z', '*') for this quantity match the same dimensions as the specified quantity.

Returns **out_dict** – A dictionary whose keys are quantity names and values are numpy arrays containing the data for those quantities, as specified by property_dictionary.

Return type dict

Raises

- `InvalidStateError` – If a DataArray in the state is missing an explicitly-specified dimension defined in its properties (dimension names other than 'x', 'y', 'z', or '*'), or if the state is missing a required quantity.
- `InvalidPropertyError` – If a quantity in property_dictionary is missing values for "dims" or "units".

`sympl.`**`restore_data_arrays_with_properties`**(*raw_arrays*, *output_properties*, *input_state*, *input_properties*)

Parameters

- **`raw_arrays`** (*dict*) – A dictionary whose keys are quantity names and values are numpy arrays containing the data for those quantities.
- **`output_properties`** (*dict*) – A dictionary whose keys are quantity names and values are dictionaries with properties for those quantities. The property "dims_like" must be present, and specifies an input quantity that the dimensions of the output quantity should be like. All other properties are included as attributes on the output DataArray for that quantity, including "units" which is required.
- **`input_state`** (*dict*) – A state dictionary that was used as input to a component for which DataArrays are being restored.

- **input_properties** (*dict*) – A dictionary whose keys are quantity names and values are dictionaries with input properties for those quantities. The property "dims" must be present, indicating the dimensions that the quantity was transformed to when taken as input to a component.

Returns **out_dict** – A dictionary whose keys are quantities and values are DataArrays corresponding to those quantities, with data, shapes and attributes determined from the inputs to this function.

Return type dict

Raises `InvalidPropertyDictError` – When an output property is specified to have dims_like an input property, but the arrays for the two properties have incompatible shapes.

1.12.2 Aliases

Note: Using aliases isn't necessary, but it may make your code easier to read if you have long quantity names

Let's say if instead of the properties we set before, we have

```
input_properties = {
    'air_temperature': {
        'dims': ['*'],
        'units': 'degK',
        'alias': 'T',
    },
    'eastward_wind': {
        'dims': ['*'],
        'units': 'm/s',
        'match_dims_like': ['air_temperature']
        'alias': 'u',
    }
}
```

The difference here is we've set 'T' and 'u' to be *aliases* for 'air_temperature' and 'eastward_wind'. What does that mean? Well, in the computational code, we can write:

```
def __call__(self, state):
    # we get numpy arrays with specifications from input_properties
    raw_arrays = get_numpy_arrays_with_properties(
        state, self.input_properties)
    T = raw_arrays['T']
    # here the actual computation happens
    raw_tendencies = {
        'T': (T - self._T0)/self._tau,
    }
    # now we re-format the data in a way the host model can use
    diagnostics = {}
    tendencies = restore_data_arrays_with_properties(
        raw_tendencies, self.tendency_properties,
        state, self.input_properties)
    return diagnostics, tendencies
```

Instead of using 'air_temperature' in the raw_arrays and raw_tendencies dictionaries, we can use 'T'. This doesn't matter much for a name as short as air_temperature, but it might matter for longer names like 'correlation_of_eastward_wind_and_liquid_water_potential_temperature_on_interface_levels'.

Also notice that even though the alias is set in input_properties, it is also used when restoring DataArrays. If there is an output that is not also an input, the alias could instead be set in `diagnostic_properties`, `tendency_properties`, or `output_properties`, wherever is relevant.

1.13 Memory Management

Warning: This section contains fairly advanced topics. If you find it confusing, that's because the behavior *is* confusing.

1.13.1 Arrays

If possible, you should try to be aware of when there are two code references to the same in-memory array. This can help avoid some common bugs. Let's start with an example. Say you create a ConstantPrognostic object like so:

```
>>> import numpy as np
>>> from sympl import ConstantPrognostic, DataArray
>>> array = DataArray(
        np.ones((5, 5, 10)),
        dims=('lon', 'lat', 'lev'), attrs={'units': 'K/s'})
>>> tendencies = {'air_temperature': array}
>>> prognostic = ConstantPrognostic(tendencies)
```

This is all fine so far. But it's important to know that now `array` is the same array stored inside `prognostic`:

```
>>> out_tendencies, out_diagnostics = prognostic({})
>>> out_tendencies['air_temperature'] is array  # same place in memory
True
```

So if you were to modify `array`, it would *change the output given by prognostic*:

```
>>> array[:] = array * 5.
>>> out_tendencies, out_diagnostics = prognostic({})
>>> out_tendencies['air_temperature'] is array
True
>>> np.all(out_tendencies['air_temperature'].values == array.values)
True
```

When in doubt, assume that any array you put into a component when it is initialized should not be modified any more, unless changing the values in the component is intentional.

However, this code would not modify the array in `prognostic`:

```
>>> array = array * 5.
>>> out_tendencies, out_diagnostics = prognostic({})
>>> out_tendencies['air_temperature'] is array
False
>>> np.all(out_tendencies['air_temperature'].values == array.values)
False
```

What's the difference? We took away the `[:]` on the left hand side of the assignment operator. when `[:]` is included, python modifies the array on the left hand side, but when it's not included it tells the python variable name "array" to refer to what is on the right hand side. These are subtly different things - one involves modifying the memory that `array` already refers to, the other involves telling `array` to refer to a different place in memory. More precisely,

having `array =` tells python that you want to change what the variable `array` refers to, and set it to be the thing on the right hand side, while `array[:] =` tells python to call the `__setitem__(key, value)` method of `array` with the contents of the square parentheses as the key and the right hand side as the value.

Interestingly, `array = array * 5.` has different behavior from `array *= 5.`. The first one will change what `array` refers to, as before, while the second one will modify `array` in-place without changing the reference. Writing `array *= 5` is the same as writing `array[:] = array * 5'`. All similarly written operations (`-=`, `+=`, `/=`, etc.) are in-place operations.

1.14 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps. You can contribute in many ways:

1.14.1 Types of Contributions

Usage in Publications

If you use Sympl to perform research, your publication is a valuable resource for others looking to learn the ways they can leverage Sympl's capabilities. If you have used Sympl in a publication, please let us know so we can add it to the list.

Working on projects that use Sympl

Sympl is only as useful as the components it has available. You can make Sympl more useful for others by contributing to model projects which use Sympl, or by writing/wrapping model components and deploying them in your own Python packages.

Presenting Sympl to Others

Sympl is meant to be an accessible, community-driven tool. You can help the community of users grow and be more effective in many ways, such as:

- Running a workshop
- Offering to be a resource for others to ask questions
- Presenting research that uses Sympl

If you or someone you know is contributing to the Sympl community by presenting it or assisting others with the model, please let us know so we can add that person to the contributors list.

Report Bugs

Report bugs at https://github.com/mcgibbon/sympl/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

Sympl could always use more documentation. You could:

- Clean up or add to the official Sympl docs and docstrings.
- Write useful and clear examples that are missing from the examples folder.
- Create a Jupyter notebook that uses Sympl and share it with others.
- Prepare reproducible model scripts to distribute with a paper using Sympl.
- Anything else that communicates useful information about Sympl.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mcgibbon/sympl/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.14.2 Get Started!

Ready to contribute? Here's how to set up *sympl* for local development.

1. Fork the *sympl* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sympl.git
```

3. Install your local copy in development mode:

```
$ cd sympl/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 sympl tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

1.14.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4 and 3.5. Check https://travis-ci.org/mcgibbon/sympl/pull_requests and make sure that the tests pass for all supported Python versions.

1.14.4 Style

In the Sympl code, we follow PEP 8 style guidelines (tested by flake8). You can test style by running "tox -e flake8" from the root directory of the repository. There are some exceptions to PEP 8:

- All lines should be shorter than 80 characters. However, lines longer than this are permissible if this increases readability (particularly for lines representing complicated equations).
- Space should be assigned around arithmetic operators in a way that maximizes readability. For some cases, this may mean not including whitespace around certain operations to make the separation of terms clearer, e.g. "Cp*T + g*z + Lv*q".
- While state dictionary keys are full and verbose, within components they may be assigned to shorter names if it makes the code clearer.
- We can take advantage of known scientific abbreviations for quantities within components (e.g. "T" for "air_temperature") even thought they do not follow pothole_case.

1.14.5 Tips

To run a subset of tests:

```
$ py.test tests.test_timestepping
```

1.15 Credits

1.15.1 Development Lead

- Jeremy McGibbon <mcgibbon@uw.edu>

1.15.2 Contributors

- Joy Monteiro <joy.monteiro@misu.su.se>

CHAPTER 2

License

sympl is available under the open source BSD License.

Index

Symbols

A

C

D

G

I

L

M

N

O

P

R

S

T

W