Letter to the Editor:
Changes in the manuscript

The manuscript has been overhauled completely to meet the comments of the reviewers. In particular, the second section has been changed completely, and now presents a series of examples which show how to traverse a model hierarchy using Sympl and Climt. We then move on to design considerations and decisions (now a single section created by merging the previous design considerations and design decisions sections), where we frequently refer to the examples in the previous section to illustrate the issues we are trying to solve using these packages. These examples eliminated the need to present an example at the end of the manuscript.

The other major change was that Sympl and Climt were updated during the review process, and this is reflected in the names of the Sympl classes and version numbers in the paper title. Some minor API changes were also documented.

Furthermore, the line-by-line comments of the reviewers for the previous version of the manuscript have been addressed before any changes were made.

Other changes that were made to the manuscript are:

- Zenodo DOIs were added for both packages.
- The figure captions were overhauled for most figures.
- The ordering in Figure 6 was changed.
- Two panels were removed from Figure 8 since they did not add much information.
- The scripts used to run the benchmark simulations have been added as supplementary material

Sincerely,
Joy, Jeremy and Rodrigo

`sympl` (v. ~~0.3.2~~0.4.0) and `climt` (v. ~~0.11.0~~0.15.3) – Towards a flexible framework for building model hierarchies in Python

Joy Merwin Monteiro[1], Jeremy McGibbon[2], and Rodrigo Caballero[1]

[1]Department of Meteorology, Stockholm University, SE-106 91 Stockholm, Sweden
[2] 408 Atmospheric Sciences–Geophysics (ATG) Building, University of Washington at Seattle, Box 351640, Seattle, Washington 98195-1640

Correspondence: Joy Merwin Monteiro (joy.merwin@gmail.com)

Abstract. `sympl` (System for Modelling Planets) and `climt` (Climate Modelling and ~~diagnostics Toolkit) represent~~ Diagnostics Toolkit) are an attempt to rethink climate modelling frameworks from the ground up. The aim is to use expressive data structures available in the scientific Python ecosystem along with best practices in software design to ~~build models that are self-documenting, highly inter-operable and that provide fine grained control over model components and behaviour. We believe that such an approach towards building models is essential to~~ allow scientists to easily and reliably combine model components to represent the climate system at a desired level of complexity, and to enable users to fully understand what the model is doing.

`sympl` is a framework which formulates the model in terms of a ~~"state" which~~ "state" that gets evolved forward in time ~~by `TimeStepper` and Implicit components, and which can be modified by `Diagnostic` components~~ or modified within a specific time by well-defined components. `sympl`'s design facilitates building models that are self-documenting, highly inter-operable, and that provide fine-grained control over model components and behaviour. ~~`TimeStepper`~~`sympl` components ~~in turn rely on `Prognostic` components to compute tendencies. Components contain all the~~ contain all relevant information about the ~~kinds of inputs~~ input they expect and ~~outputs~~ output that they provide. Components ~~can be used interchangeably~~are designed to be easily interchanged, even when they rely on different units or array configurations. `sympl` provides basic functions and objects which could be used ~~by~~ in any type of Earth system model.

`climt` is an Earth system modelling toolkit that contains scientific components built ~~over the~~ using `sympl` base objects. ~~Components can be written in any language accessible from Python , and Fortran/C librariesare accessed via Cython~~These include both pure Python components and wrapped Fortran libraries. `climt` ~~aims to provide different user APIs which trade-off simplicity of use against flexibility of model building,~~ provides functionality requiring model-specific assumptions, such as state initialisation and grid configuration. `climt`'s programming interface designed to be easy to use and thus appealing to a wide audience.

Model building, configuration and execution ~~is~~ are performed through a Python script (or Jupyter Notebook), enabling researchers to build an end-to-end ~~Python based~~ Python-based pipeline along with popular Python ~~based data analysis tools. Because of the modularity of the individual components, using online data analysis, visualisation or assimilation algorithms and tools with `sympl`/`climt` components is extremely simple.~~ data analysis and visualisation tools.

1 Introduction

The climate is a complex system ~~constituted of a heterogeneous set of mutually~~ composed of interacting subsystems (atmosphere, ocean, cryosphere, biosphere, chemosphere), each encompassing a broad range of physical and chemical processes with space and time scales spanning many orders of magnitude. Modelling and understanding the climate system in its entirety is ~~truly~~ a grand scientific and technological challenge. An increasingly recognised strategy for tackling this challenge is to build a hierarchy of models of varying complexity. Simpler models are more amenable to in-depth analysis and understanding; the insight gained from these simpler models can then be used to understand ~~slightly~~ more complicated models, and so on (Held, 2005). Specifying which particular models should occupy each rung in such a hierarchy is necessarily a matter of subjective choice, and the questions of how to create a hierarchy and what models are desirable in a canonical hierarchy has generated extensive discussions (Jeevanjee et al., 2017). Our purpose here is to present a modelling framework which enables climate scientists to easily and transparently traverse the ~~modelling hierarchy~~ specific model hierarchy suiting their needs.

Designing and building a framework that facilitates traversing this hierarchy ~~is non-trivial and continues to remain~~ remains a challenge. The lack of flexibility of existing climate models forces scientists to spend a lot of time reading and modifying code to construct alternative model versions that should in principle be straightforward to build. Most modelling frameworks simply provide code to exchange information between different physical domains such as atmosphere and ocean (See Theurich et al., 2015; Valcke et al., 2012, for a survey of modelling frameworks), with each physical domain being represented by a monolithic block of code. It was only with the advent a new generation of frameworks like the Earth System Modelling Framework (ESMF) (DeLuca et al., 2012; Theurich et al., 2015) that fine-grained control over the components that constitute a climate model was made possible. For instance, ESMF allows configuring components as trees, where the leaf nodes could represent physical processes such as radiation and the root of the tree could represent a physical domain such as the atmosphere. Such a hierarchical ordering of physical processes is also present in Python based modelling packages – previous versions of `climt` and `climlab` (https://github.com/brian-rose/climlab) allow building models in a manner similar to ESMF. However, the norm continues to be that climate ~~models are~~ model composition is configured by namelist variables and boolean flags in the code rather than ~~framework based~~ framework-based approaches (like the component trees that ESMF allows). ~~`sympl` and `climt`, on the other hand, allow finer grained control over the composition of the model, with individual components representing physical processes (such as radiation, convection, etc.,) rather than physical domains. Attempting to model the climate system at this fine-grained level has its own challenges, which we attempt to solve in these packages. Initiatives to build frameworks to traverse the hierarchy between highly idealised models to full scale GCMs (general circulation models) do exist (Fraedrich et al., 2005; Vallis et al., 2017), but we believe the fine-grained configuration allowed by our design to be unique.~~

Another emerging concern in the scientific community is that of reproducibility of research (Peng, 2011). While publicly available climate models do provide validated configurations that are in principle completely reproducible, climate scientists routinely need to make changes to the code that are hard to document (or understand)~~, as mentioned above~~. Short of sharing a copy of the entire code base, such modifications makes it difficult to reproduce simulations. While some level of code

manipulation is inescapable, we note from our own experience and from reading about such modifications in the literature that most of them follow set patterns which could easily be provided by modelling frameworks themselves.

In this paper we present a new modelling framework, `sympl`, and a model toolkit, `climt`. While `sympl` focuses on providing a model framework, a rich taxonomy of components, and model agnostic configuration options, `climt` focuses on providing a broad array of physical components to allow users to build scientifically useful models. `climt` also provides model dependent configuration options and helper functions to create an initial model state as required by the components.

`sympl` and `climt` ~~were written to address some of these issues, and to create a modelling framework that is~~ allow finer grained control over the composition of the model, with individual components representing physical processes (such as radiation, convection, etc.,) rather than physical domains. Attempting to model the climate system at the physical process level has its own challenges which we attempt to solve in these packages. Initiatives to build frameworks to traverse the hierarchy between highly idealised models to full scale GCMs (general circulation models) do exist (Fraedrich et al., 2005; Vallis et al., 2017), but we believe the definition of a clear set of classes to represent the physical process level of the model to be unique.

`sympl` and `climt` allow writing models which are easy to use and ~~facilitates~~ facilitate reproducibility of simulations. Both these packages are subject to ongoing development, but have reached a level of maturity that makes it worthwhile to document them here. In ~~the following sections,~~ section 2, present a series of models written using `sympl` and `climt` which illustrate the construction of a model hierarchy. In section 3 we describe some ~~design issues~~ challenges that modelling frameworks have to solve ~~, followed by the design of~~ (in the context of the above examples when possible) and discuss how `sympl` and `climt` ~~, some example scripts and~~ address these challenges. We then discuss in more detail the interfaces of `sympl` (section 5) and `climt` (section 6). In section 7 we present some benchmark calculations, and conclude with a discussion of developments planned in the future.

2 ~~Climate modelling frameworks – general considerations~~`sympl` and `climt` in action

~~For the purposes of this paper, we define a modelling framework as a library or libraries that allow~~ To illustrate the advantages of the ~~creation of climate models by providing abstractions of "infrastructure" code. This infrastructure could include model-independent parts: data, time~~ fine-grained control that `sympl` and ~~units management, component description and interoperability as wellas model dependent parts: component configuration, model state and grid initialisation and model creation from components .~~ `climt` offer, we consider a series of examples starting with a diagnostic radiative calculation and ending with an idealised three dimensional atmospheric general circulation model.

~~Climate modelling frameworks should make it possible to combine arbitrary model components in an intuitive manner to create a model of complexity appropriate to the scientific question at hand~~ Figure 1 shows the script required to calculate the heating tendencies from a longwave radiative transfer component (Clough et al., 2005). A detailed explanation of the script follows:

- **Line 1**: Import the `climt` package.
- **Line 4**: Instantiate a radiative transfer component.

```
1  import climt
2
3  # Create component
4  radiation = climt.RRTMGLongwave()
5
6  # Create model state
7  model_state = climt.get_default_state([radiation])
8
9  tendencies, diagnostics = radiation(model_state)
```

Figure 1. A Python script which calculates the heating tendencies and associated diagnostics from a longwave radiative transfer component. See the text for a detailed description.

- **Line 7**: Create a state dictionary which contains all the quantities required as inputs by the radiative transfer component.

- **Line 9**: Calculate the heating rate (available in `tendencies` and any associated diagnostics such as the radiative fluxes (available in `diagnostics`).

In Figure 1, we have used the default values for quantities in `model_state`, which corresponds to an isothermal atmosphere.

This example, though seemingly simple, is remarkable because of the ease with which such a diagnostic calculation can be performed. Traditionally, such a calculation would involve writing a Fortran "driver", compiling it with the radiative transfer library, writing the output to a file, and then reading the output file into a suitable environment for further analysis. The ease of use illustrated in the above example is a direct result of the fine-grained control that `sympl` and `climt` allow – ~~ideally, the user simply has to specify the components desired,~~ individual components can be configured and run (interactively, if desired) without having to compile them with a driver file. To summarise, *components*, not models, are first class entities within the `sympl` framework.

It is worth examining this example in greater detail, since it highlights some important features of `sympl` and `climt` that we will look at closely in subsequent sections. The component called `radiation` is an implementation of the `sympl` entity called `TendencyComponent`, which is a template for components which calculate tendencies of a quantity (air temperature in this case) based on quantities in a state dictionary. We will encounter other kinds of components in subsequent examples.

Figure 2 builds upon the previous example to create a model that includes both radiation and convection, steps the state quantities forward in time, and writes the output to a file. The changes from the previous script are as follows:

- **Line 7**: Define the model time step using `timedelta` from the `datetime` library, which is part of any Python distribution.

```
from sympl import (
        AdamsBashforth, NetCDFMonitor)
import climt
from datetime import timedelta

# Define model timestep in minutes
model_timestep = timedelta(minutes=1)

# Create components
radiation = climt.RRTMGLongwave()
convection = climt.EmanuelConvection()
boundary_layer = climt.SimplePhysics()

# Create model state
model_state = climt.get_default_state(
    [radiation, convection, boundary_layer])

# Create integrator
time_stepper = AdamsBashforth(
    [radiation, convection])

# Create monitor
monitor = NetCDFMonitor('radiative_convective.nc')

# step model forward
for step in range(10):
    bl_diagnostics, bl_new_state = boundary_layer(
        model_state, model_timestep)
    model_state.update(bl_diagnostics)
    model_state.update(bl_new_state)

    diagnostics, new_state = time_stepper(
        model_state, model_timestep)
    model_state.update(diagnostics)
    monitor.store(model_state)
    model_state.update(new_state)
    model_state['time'] += model_timestep
```

Figure 2. A Python script which calculates the radiative-convective equilibrium temperature of an atmospheric column for a fixed surface temperature. See the text for a detailed description.

- **Line 21**: Create a time integrator which steps the model state forward in time, using tendencies generated by the radiation and convection components. The integrator chosen is an instance of the ~~framework should be able to automatically build the model~~`sympl.Stepper` class which implements a variety of Adams-Bashforth schemes.

- **Line 24**: Create a "monitor" component which writes the model state to a netCDF file.

- **Lines 27-37**: The boundary layer component provides new values of model quantities, which are used to update the model state. The time integrator incorporates the tendencies due to radiation and convection and provides new values for the model state as well. The current model state is updated with diagnostics and written to a netCDF file. The model state is then updated with the new model quantities to prepare for the next iteration.

This example illustrates how to piece together a radiative-convective equilibrium (RCE) model from different kinds of `sympl` components. This example also moves up the model hierarchy, away from static diagnostic calculations to a model which evolves in time. It is worth noting that the first half of the example which creates components and a model state remains identical to the procedure followed in the previous example. This example shows two notable features of the design of `sympl` and `climt` – One, individual components can step the model state forward themselves and Two, the model integrator is a separate entity in itself which can be replaced easily (to use more stable integration schemes, for example). These features are in keeping with our goal of capturing the diversity of model components – some of which produce tendencies, and others that produce new state quantities – and allowing the user fine-grained control over model configuration.

This example also illustrates how `sympl`'s design provides a clear understanding of the model to users. Configuring the model consists of modifying a run script which is meant to be entirely legible to the user. By reading the run script, one can see exactly which model components are being used, how the state is initialised, what configuration options are being passed to which components, what time integration scheme is used on which components, the order in which components are being called, and the point within the integration where the state is being output to a file.

Figure 3 presents the next step in the hierarchy from Fig. 2, creating a moist atmospheric general circulation model (AGCM) in an aqua-planet configuration with a prescribed sea surface temperature. The default surface temperature is horizontally uniform, but we omit prescribing a realistic temperature distribution for purposes of comparison with Fig. 2. The main differences from the previous example are:

- **Line 14**: The boundary layer component is wrapped to output tendencies instead of a new state. This wrapper is required since the spectral dynamical core must apply tendencies to most quantities in spectral space for numerical reasons.

- **Line 18**: The spectral dynamical core is used as the time stepper instead of the Adams-Bashforth scheme used previously.

- **Lines 22 and 25**: Since the model now has three dimensions, we require a grid describing the latitudes, longitudes as well. Line 18 creates a model grid, and the default model state created in Line 21 uses this grid to create an appropriate three dimensional state.

```python
from sympl import (
       TimeDifferencingWrapper, NetCDFMonitor)
import climt
from datetime import timedelta

# Define model timestep in minutes
model_timestep = timedelta(minutes=1)

# Create Components
radiation = climt.RRTMGLongwave()
convection = climt.EmanuelConvection()
boundary_layer = TimeDifferencingWrapper(
    climt.SimplePhysics())
time_stepper = GFSDynamicalCore(
    [radiation, convection, boundary_layer])

# Create model grid
model_grid = climt.get_grid(nx=64, ny=64, nz=28)

# Create model state
model_state = climt.get_default_state(
    [time_stepper], grid_state=model_grid)

# Create monitor
monitor = NetCDFMonitor('moist_agcm.nc')

# step model forward
for step in range(10):
    diagnostics, new_state = time_stepper(
        model_state, model_timestep)
    model_state.update(diagnostics)
    monitor.store(model_state)
    model_state.update(new_state)
    model_state['time'] += model_timestep
```

Figure 3. A Python script which creates an idealised moist atmospheric general circulation model. See text for a detailed description.

A remarkable fact about this example is that it is only one line longer than the previous example. Intuitively, this seems appropriate – an AGCM can be thought of as a collection of radiative-convective columns which communicate with each other via the dynamical core. However, in most modelling frameworks the intuitive picture of the transition from an RCE to an AGCM does not easily translate to code. This plug-and-play behaviour is a direct consequence of the modularity of individual `sympl` components and the fact that all components of a similar kind (`AdamsBashforth` and `GFSDynamicalCore` in this case) implement the same interface, making it possible to reuse almost the entire model script from the RCE case. In this way, `sympl` and `climt` allow constructing models in such a way that a change that intuitively seems small also translates to a code change that is small.

In this following section, we look at detail at the design decisions that allow for the construction of a model hierarchy as described in the three examples in this section.

3 Design considerations and choices

In this paper we distinguish between modelling frameworks, model toolkits, and models themselves. A framework (such as `sympl`) consists of abstractions of "infrastructure" code that allow the creation of climate models. A framework creates rules one must follow, but by doing so ensures models using the framework are easier to write, understand, and combine with one another. A toolkit (such as `climt`) implements those abstractions for concrete physical processes, and may provide additional functionality that is not covered by the framework. A model itself (such as in Figures 2 and 3) is written using components that may come from a toolkit which follows the guidelines of the framework.

Modelling frameworks should enable scientists to intuitively combine model components and create an appropriately complex model for the scientific question at hand. The user should also be able to specify details such as the order in which components are called and the time stepping schemes used. ~~In addition, it~~ Model toolkits should provide a wide variety of components that enable users to write a model appropriate to the question at hand. Toolkits should also maintain a list of quantities and numerical grids that are required by the components it provides to facilitate creating model arrays. It is also desirable that the process of creating a model is fairly easy to understand ~~,~~ and that the model code be self-documenting to eliminate the need to write additional documentation whenever possible. ~~To achieve these goals, the following challenges must be addressed by any modelling framework.~~

3.1 ~~Taxonomy of model components~~

~~Typically~~

3.1 Component diversity

One of the major aims of `sympl` and `climt` is to allow fine-grained control over the processes that constitute an Earth system model. In currently available modelling frameworks, the integration of scientific code (or model components) and the modelling framework happens at the physical domain level – atmosphere, ocean, land and so on. The processes ~~the~~ that operate

within each physical domain (fluid dynamics, radiation, convection etc.,) are not accessible in a systematic manner, and their ~~integration into the model code is harder to unravel~~code becomes tightly coupled and difficult to modify. For example, changing the radiation code in an ~~atmosphere~~ atmospheric model is typically harder than changing the kind of ocean the atmosphere is coupled to (slab, dynamical, etc.,). ~~This tight integration~~ Tight coupling of code at the process level can make these models ~~highly efficientbut less flexible to work with.~~ more efficient, but at the cost of scientist efficiency, as the code is significantly less flexible and have undocumented and complicated interfaces between components.

Furthermore, the fact that certain process level components are written to work only with certain other components demands an "architectural unity" (Randall, 1996) which might also encourage tight integration of model components. Within `sympl`, it may still be the case that two components are incompatible with one another (for example, using different thermodynamic quantities), but because their interfaces are clearly defined, it is easier to couple these components (for example, by converting thermodynamic quantities using an additional component). Other incompatibilities are handled automatically. For example, `sympl` automatically ensures components which use different units or dimension orderings can be used alongside one another.

If we allow for configuration at the process level, we are then faced with model components which behave quite differently: Some components (like radiation) return tendencies, while others (like large scale condensation) return a new value of a physical quantity. ~~Any modelling framework must therefore have an ontology of components that is sufficient to capture this diversity, especially if the goal is to automate model construction~~`sympl` provides a set of component classes that is comprehensive enough to capture the diversity of component behaviours.

3.2 ~~Behaviour of model components~~

~~Model components can display a variety of behaviours which must be exposed in a consistent manner by the modelling framework. For instance, model components could Normally return a new value rather than a tendency of some physical quantity, but in certain instances a tendency might be desirable. Provide output that is piece-wise constant in time – the output is updated only once every N iterations,~~ An example of a `sympl` component is depicted in Fig. 4. `input_properties` and ~~the same value it output for the next $N - 1$ iterations. This behaviour is normally used in radiative transfer codes. Scale some of its inputs or outputs by some floating point number. This kind of behaviour is desirable for examplein mechanism denial studies.~~ `tendency_properties` are 'property' attributes containing details of the required inputs and returned outputs, tendencies or diagnostics. Here, the input is a quantity `air_temperature` whose horizontal dimensions are `latitude`, `longitude` and whose values in the vertical are defined at model mid-levels. The units are specified for each quantity. The `array_call` method will be called by `sympl` with numpy arrays that are automatically extracted from the model state to satisfy the input property specifications. It is written to return numpy arrays as output which satisfy its tendency and diagnostic property specifications. The `__call__` method which is called, for example, in Line 9 of Fig. 1, is implemented by the base `sympl` classes. This method encapsulates the boilerplate code of performing consistency checks on the model state, extracting numpy arrays with the correct units and shape, and creating a new state dictionary from the arrays returned by `array_call`.

```
lass PrescribedHeating(TendencyComponent):

  input_properties = {
      'latitude': {
          'dims': ['*'],
          'units': 'degrees_N',
      },
      'longitude': {
          'dims': ['*'],
          'units': 'degrees_E',
      },
      'air_pressure': {
          'dims': ['mid_levels', '*'],
          'units': 'Pa',
      },
  }

  diagnostic_properties = {}

  tendency_properties = {
      'air_temperature': {
          'dims': ['mid_levels', '*'],
          'units': 'degK s^-1',
      }
  }

  def __init__(self, forcing_filename, **kwargs):
      [...]
      super(PrescribedHeating, self).__init__(**kwargs)

  def array_call(self, state):
      [...]
```

Figure 4. The general code layout for a `sympl` component.

~~We can modify certain behaviours of model components, such as the ones above, by interacting with only the inputsand outputs of scientific components. Such modifications can be applied in the same manner to different components~~Dimension and unit requirements in `sympl` are not restrictions on the inputs, but rather describe the internal representation used by the component. `sympl` will automatically convert the input state to satisfy these requirements, and raise an exception if that is not possible. In this way, the property dictionaries act as self-documenting code, which both documents the component interface and is used to convert input arrays to the desired dimension ordering and units for the component.

As a `TendencyComponent` (discussed in Section 5), this component outputs tendencies of a quantity `air_temperature`. The `array_call()` method accepts a dictionary containing just the `numpy` arrays extracted from the model state with the correct units and dimensions and returns the temperature tendency as specified in the component properties.

4 ~~Configuration of models~~

~~Models need to be configured at multiple levels,~~ Since components in `sympl` and ~~modelling frameworks have to provide~~ `climt` are first class entities, they are not dependent on any other code for execution – Fig. 1 shows how to interact solely with a radiative component without the need for an integrator or any other entity. This behaviour serves an important educational purpose and facilitates diagnostic calculations made very often during research.

3.1 Model configuration

Together, `sympl` and `climt` allow a natural way of configuring ~~the following~~ various aspects of a ~~model~~climate model listed below:

- Physical configuration (toolkit agnostic): the physical constants required by the model components.
- Algorithmic configuration (toolkit specific): the "tunable" parameters which modify the behaviour of the algorithm which represent physical processes – ~~like~~ for example, the entrainment coefficient in a convective parameterisation~~, for example~~.
- ~~Behavioural configuration : described in the previous section~~Interfacial configuration (toolkit agnostic): Modifications applied to inputs and outputs at the interface of a component, further described below.
- Memory and Computing resource configuration (toolkit specific): The layout of arrays used by the model and the distribution of the model components over the available number of processors and co-processors.
- Compositional configuration (toolkit specific): The components that compose a model, any dependency between components, the order in which components are executed all need to be ~~quantified. While choosing the order of components, it must be kept in mind that the ordering of components can have an impact on the simulated climate of the model (Donahue and Caldwell, 2018).~~ described.

3.2 ~~Interacting with model components~~

~~It would be desirable to be able to interact with individual components at runtime – Not only would this serve an important educational purpose, but it would also facilitate diagnostic calculations made very often during research. For instance, it would be very helpful if one could calculate radiative tendencies without having to build a full model to access the radiative transfer component.~~

3.2 ~~Combining model components~~

~~Different model components (especially those from different physical domains) can use different discretisations of the domain, or model grids. Combining components which use different grids involves interpolating from one grid to another in the case one of these components requires information provided by other components. This functionality is provided by a separate component known as the coupler (Valcke et al., 2012). It would be desirable that the modelling framework is able to identify that the constituent components of a model are on different grids and provide support for adding couplers when necessary.~~

3.2 ~~Models and computing infrastructure~~

~~Models need to run on a variety of computing facilities, and modelling frameworks need to abstract away the details of the computing infrastructure to enable model users to focus on the scientific aspects of the simulations. Creating such an abstraction is challenging due to the introduction of new architectures and computing models such as Graphics Processing Units (GPUs) and many-core CPUs (See introduction in Balaji et al., 2016). The modelling framework must facilitate storage of information about the computational resources available so that components that can take advantage of certain specialised hardware will be able to do so.~~

Figure 5 shows ~~the wide variety~~ types of configuration information and ~~the places~~ where such configuration lies ~~in~~ `climt`/`sympl` and in traditional climate models. ~~The~~ In contrast to `sympl` where all configuration passes through a readable run script, the sheer variety of locations where the configuration resides in traditional models makes it hard to keep track of what configuration information has changed~~and~~ , and how configuration changes affect model runs. This makes it daunting for beginners to ~~setup models. In contrast, model~~ write models. Model configuration using `sympl` and `climt` is highly centralised and easily accessible - all configuration passes through the model run script, which is written to be readable and accessible to model users. Such centralised configuration ~~also~~ reduces errors arising due to misconfiguration of the model.

~~The variety of configuration options in a climate model and in the `sympl-climt` framework. Note that not every climate model uses all configuration options. The starred boxes in the `sympl+climt` side indicate functionality that is not yet implemented.~~

4 ~~Design Decisions~~

During the development of the framework described in this paper, it was realised that some aspects of model development were fully model agnostic, whereas some aspects such as model state initialisation, ordering of coordinates axes and labelling conventions were best decided by each model. The model agnostic parts were included in the framework package (`sympl`) whereas the model specific parts were included in the model package (`climt`).

Python was used as the language to write the framework. Python as a language and the Python ecosystem have a number of desirable features, all of which were taken advantage of during the development of `sympl` While most of the configuration elements listed above are familiar, interfacial configuration is mostly unheard of and `climt`: usually applied in a non-systematic manner within climate models. The `TimeDifferencingWrapper` used in line 14 of Figure 3 is an example of interfacial configuration of a climate model component. There are a variety of interfacial modifications which are commonly applied to components, and can be applied in a consistent manner across components. For instance, model components could

- Earlier versions of `climt` were written in Python, which gave the authors an idea of the convenience and flexibility it afforded. In particular, the object orientation capabilities of Python provide a straightforward way to represent the component based architecture adopted by almost all climate modelling frameworksNormally return a new value rather than a tendency of some physical quantity, but in certain instances a tendency might be desirable (as in Figure 3).
- Scientific libraries within the Python ecosystem now offer acceptable performance for computationally intensive operations typically used in climate modelsProvide output that is piece-wise constant in time – the output is updated only once every N iterations, and the same value it output for the next $N-1$ iterations. This behaviour is normally used in radiative transfer codes.
- The Python ecosystem includes a wide variety of libraries that might be useful for climate models. Examples include machine learning, graphics and web services libraries. Python's ability to act as a glue language allows interfacing with the large number of libraries for climate modelling already available in Fortran. Tools available in the Python ecosystem like jupyter, `pytest` and `sphinx` which enable writing reproducible workflows and code that is well documented and tested. Scale some of its inputs or outputs by some floating point number. This kind of behaviour is desirable for example in mechanism denial studies.

We can interfacially modify certain behaviours of model components, such as the ones above, by interacting with only the inputs and outputs of scientific components. `sympl` formalises such configuration by providing wrapper objects like `TimeDifferencingWrapper`.

4 Model arrays

Model arrays are contained within the `DataArray` abstraction provided by `xarray` [1] (Hoyer and Hamman, 2017) rather than the more low-level `numpy` array. `DataArrays` are an abstraction over `numpy` arrays with a more natural fit to climate data by providing labeled dimensions and metadata storage capabilities.

[1] see xarray documentation at

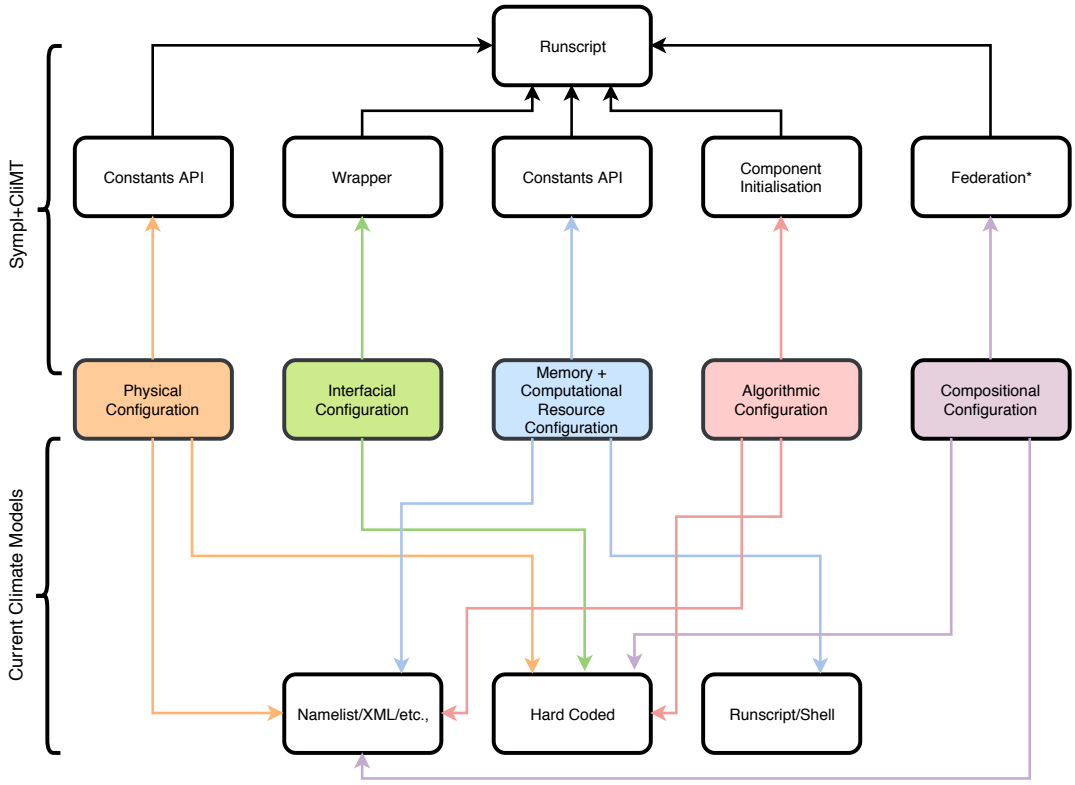

Figure 5. The variety of configuration options in a climate model and in the `sympl-climt` framework. The coloured boxes indicate the type of configuration and the white boxes parts of the model code base. Arrows from the coloured to the white boxes indicate the part of the code base that is typically responsible for the configuration from which the arrow originates. A particular type of configuration could exist in two different parts of the code base: such a situation is indicated by multiple arrows terminating at all the relevant white boxes. Note that not every climate model uses all configuration options. The starred box on the `sympl+climt` side indicates functionality that is not yet implemented.

`sympl`'s `DataArray` object is a subclass of the `xarray DataArray` that provides units handling and conversion and will be described subsequently. ~~A fortunate side-effect of this design choice is that the analysis capabilities built into~~ This exposes the powerful analysis capabilities of `xarray`~~are automatically available~~, allowing users to build an end-to-end pipeline entirely within Python, from simulation to data analysis and generation of publication-ready figures.

Since low-level array operations using `numpy` and `xarray` are fairly simple, especially changing coordinate ordering and C/Fortran memory ordering, `climt` only provides guidelines for memory layout of arrays. However, changing the ordering of dimensions in memory will incur ~~a performance penalty~~ the performance penalty of copying the array if the model ~~uses any components which wrap Fortran libraries.~~calls compiled code.

In the interest of readability of component and model code, ~~we strongly encourage using~~ `sympl` strongly encourages and `climt` makes use of descriptive names for model quantities, ~~preferably~~ adhering to the CF conventions[1] when possible. Most pre-defined quantities in `climt` use names derived from the CF conventions. One additional suffix that we found necessary to use was `on_interface_levels` to distinguish between quantities defined on the interfaces and mid-levels of the vertical grid. For example `air_temperature` refers to the air temperature defined at the vertical grid centre, whereas `air_temperature_on_interface_levels` refers to the same quantity defined at the vertical grid edges. This convention is only a requirement within the model state. Within an individual component, shorter names can be used for variables representing quantities. This shorter name is also contained in the property dictionary of the component, serving as documentation for the meaning of that shorter variable name.

4.1 Modelling language

Python was used as the language to write the framework. Python as a language and the Python ecosystem have a number of desirable features, all of which were taken advantage of during the development of `sympl` and `climt`:

- Earlier versions of `climt` were written in Python, which gave the authors an idea of the convenience and flexibility it afforded. In particular, the object-oriented capabilities of Python provide a straightforward way to represent the component based architecture adopted by almost all climate modelling frameworks.
- Scientific libraries within the Python ecosystem now offer acceptable performance for computationally intensive operations typically used in climate models.
- The Python ecosystem includes many libraries which can be useful in developing climate models. Examples include machine learning, graphics and web services libraries.
- Python's ability to act as a glue language allows interfacing with the large number of libraries for climate modelling already available in Fortran.
- Tools available in the Python ecosystem like `jupyter`, `pytest` and `sphinx` which enable writing reproducible workflows and code that is well documented and tested.

5 `sympl` – Design and programming interface

`sympl` conceives of a climate model as a state that is continuously updated by various components. `sympl`'s taxonomy consists of seven kinds of components. Four of these component types are used to represent physical processes and the remaining represent other functionality required to build and run models~~:~~.

[1] http://cfconventions.org/Data/cf-standard-names/48/build/cf-standard-name-table.html

- ~~Prognostic~~TendencyComponent ~~components~~ objects like RRTMGLongwave in Fig. 1 which take the model state as input and returns tendencies of ~~certain quantities and optional diagnostics as output~~quantities and values of quantities defined at the time of the input state.

- ~~Implicit~~Stepper components like SimplePhysics in Fig. 2 which take the model state and a timestep as input and returns ~~a new values of certain quantities and optional diagnostics as output.~~values of quantities defined at a new time (after the timestep) and optionally diagnostic values of quantities defined at the time of the input state. These are implicit as they define the target model state in terms of the target model state (e.g. that the target state is not supersaturated).

- ~~Diagnostic~~DiagnosticComponent ~~components~~ objects which take the model state as input and return ~~diagnostics~~ quantities defined at the time of the input state as output.

- ~~ImplicitPrognostic~~ImplicitTendencyComponent ~~components~~ objects like EmanuelConvection in Fig. 2 which return tendencies, but require the model timestep to produce these tendencies. This ~~requirement may be to implement flux limiting or to ensure that the tendencies satisfy the CFL criterion. This kind of behaviour is mainly used~~ is required when the tendencies are defined in terms of the target model state, as is often done in convection schemes ~~.~~or flux limiters. These should generally be written as Stepper components which can later be wrapped into ImplicitTendencyComponent objects if needed (such as with SimplePhysics in line 14 of Fig. 3).

- ~~TimeStepper~~TendencyStepper components like AdamsBashforth in Fig. 2 which contain a set of ~~Prognostic~~Tender ~~components~~ objects and use the tendencies they output to integrate the model state forward in time.

- Monitor components like NetCDFMonitor in Fig. 2 which provide a store method which takes the model state as input and "stores" it. The implementation of this method is left to the user, and currently is used ~~to implement monitors~~ for NetCDF output and plotting.

- ~~Wrapper components are a special case of other component types which~~ Wrapper components like TimeDifferencingWrappe in Fig. 3 which contain other sympl components and modify the inputs passed to or outputs generated by the 'wrapped' component.

Schematics of how the above components interact with the model state are presented in Figure 6. A DiagnosticComponent object (Panel a) is very simple, producing diagnostic quantities that are inserted or updated in the current model state. Monitor objects (Panel b) take in the model state and perform some action using it. Slightly more complicated is the Stepper object (Panel c), which steps the model state forward in time. In addition to producing a new state, it can produce diagnostic quantities which are inserted or updated in the current model state. Panel d is the most complex, depicting how TendencyComponent objects are used with a TendencyStepper to update the model state. Once created, a TendencyStepper behaves exactly like an Stepper object. Internally, it provides the input state to the TendencyComponent objects to compute tendencies, and uses those tendencies according to its time stepping scheme to evolve the model state forward in time. The TendencyStepper provides the same model state to all TendencyComponent objects it contains and sums the

tendencies before stepping forward in time (See Fig. 6d). Using other time marching algorithms such as sequential tendency or sequential update splitting (Donahue and Caldwell, 2018) will require users to implement their own `TendencyStepper` object, or to call several `Stepper` and `TendencyStepper` components in sequence.

As mentioned previously, wrapper components contain a "wrapped" component ~~used for computation and modify its behaviour as described in Sec. ??.~~ and modify the inputs or outputs, changing how the component appears to behave. Currently, `sympl` has the following wrappers:

- `TimeDifferencingWrapper` creates tendencies from the output of an ~~`Implicit`~~`Stepper` component by first order differencing. This creates an `ImplicitTendencyComponent` from an `Stepper` component, which is required when using spectral methods.
- `UpdateFrequencyWrapper` calls ~~the wrapped~~ a wrapped ~~`Prognostic`~~`TendencyComponent` only after the user-specified time interval has elapsed, and ~~simply outputs this value until the next time~~until then outputs the previously returned value. In effect, ~~it~~ this creates a piecewise constant output tendency which can reduce the computational load during a simulation. This is often used on radiation schemes.
- `ScalingWrapper` scales the inputs ~~before passing it onto~~ passed into the wrapped component and the outputs (new state, tendency or diagnostic) returned by the component. ~~`TendencyInDiagnosticsWrapper` returns all tendencies generated by a component as part of its diagnostics.~~

This ~~ontology~~ taxonomy of components is larger than ~~the ontology~~ those typically used in modelling frameworks. For example, ESMF only considers two kinds of components – Gridded and Coupler components. However, as discussed previously, this ~~enlarged ontology~~ extended taxonomy is required to capture the diversity of components that arise if ~~configuration of models is done~~ models are written to be configurable and modular at the process level.

5.1 Model State and the `DataArray` abstraction

The model state is a dictionary whose keys are the names of model quantities and values are `sympl DataArray` objects. The model state also contains a required keyword `time` whose value is an object that implements the Python `datetime` or `timedelta` interface. `sympl` provides an interface to use the `datetime` objects from the `cftime`[2] package to support several different calendars, as well as dates not supported by the `numpy datetime64` or built-in `datetime` objects. A schematic of the model state is presented in Fig. 7. `sympl` does not put hard restrictions on the name of model quantities, though standardised names should be used to ensure inter-package compatibility. All `DataArray` objects must define a string attribute called `units`, which is used to convert the data contained within to the appropriate units requested by a component. The units conversion is performed internally using the `Pint`[3] library. Since the actual contents of the state is dependent on model details, `sympl` assumes that the initialisation of the model state will be done by a model package (such as `climt`, or by the user.

[2] https://github.com/Unidata/cftime

[3] https://pint.readthedocs.io/en/latest/

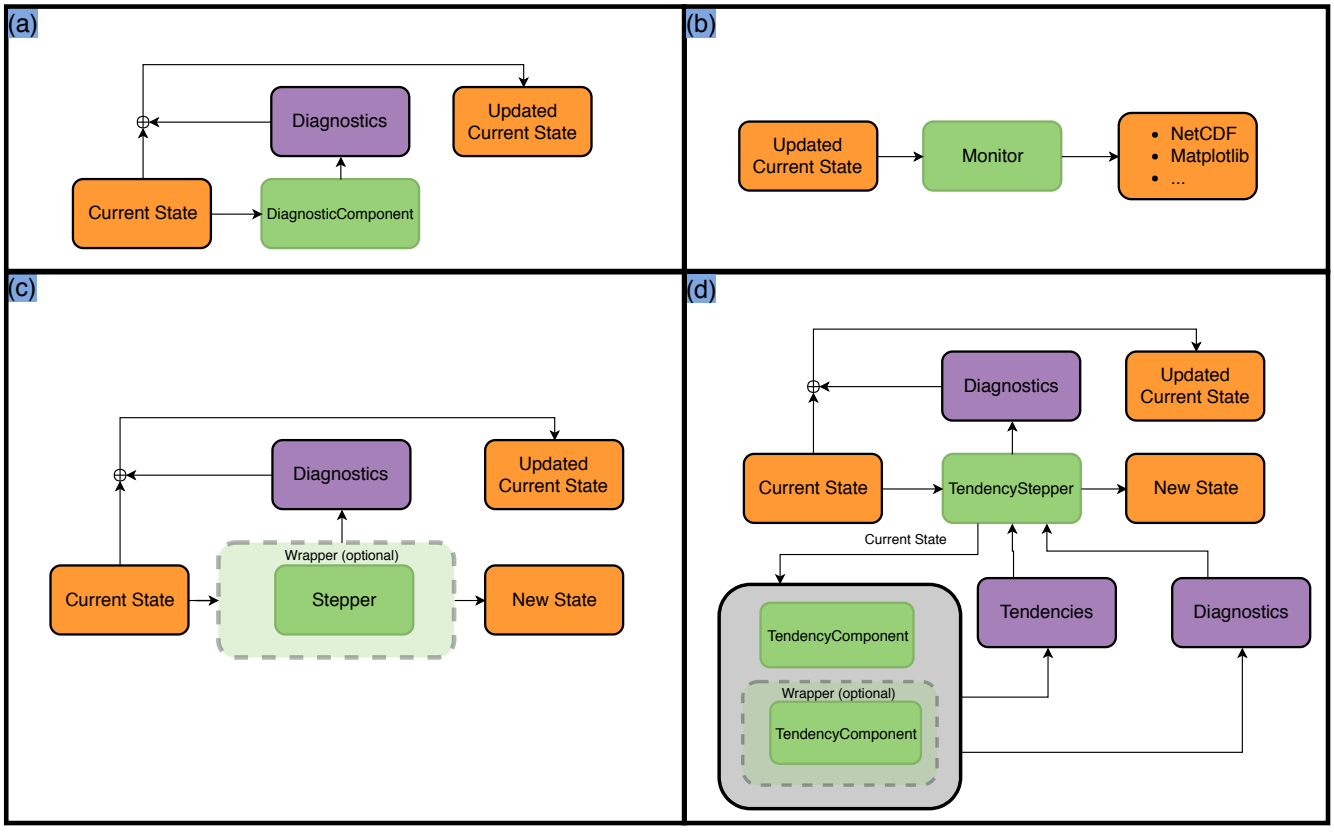

Figure 6. Flow of data for each type of component in `sympl`. The four panels show: **a)** `DiagnosticComponent`, which creates diagnostics (purple box) based on the current state. The current state is updated with the resulting diagnostics, resulting in the updated current state. **b)** `Monitor`, which can "store" the updated current state into some format like NetCDF or plots. **c)** `Stepper`, which determines a new state from the current state and a time step, and also diagnostics from the time of the current state which are used to update the current state. The current state could then be passed on to `Monitor` components (see panel **b**). **d)** `TendencyStepper`, which is a special case of `Stepper` initialised with a list of `TendencyComponent` objects (denoted by green boxes within a grey box). It passes the current state on to those `TendencyComponent` objects to compute tendencies and diagnostics, which are used to compute its outputs (generally using a time stepping scheme). In all figures, dark green boxes denote components, light green boxes denote optional wrappers, orange boxes indicate the state dictionary at different times and purple boxes indicate tendencies and diagnostics generated by components. The converging arrows at a summation symbol (plus sign inscribed within a circle) denotes updating the state dictionary (orange) to include output (purple) quantities. Examples of wrapper placements are not exhaustive and meant only as examples.

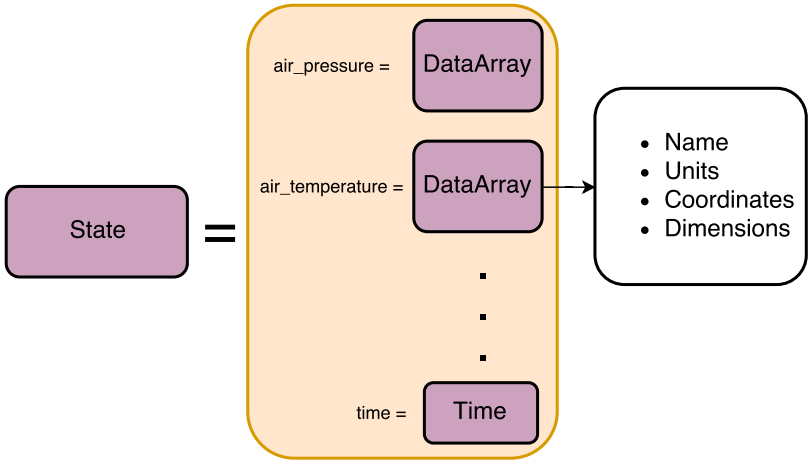

Figure 7. The model state as in the `sympl` framework. The state in the orange box contains all the information that is stepped forward in time. Each `DataArray` contains information such as the quantity name, units and dimensions/coordinates.

5.2 ~~Model dimensions~~

~~`sympl` stores the names of spatial dimensions required by model components, but treats model coordinates (e.g. latitude) as any other state quantity. Each of the dimensions `x, y, z` can have arbitrary names and new names can be added – for instance, `climt` adds two names for the vertical dimension, `mid_levels` and `interface_levels`. This facility can be used, for instance to define different names for dimensions used in land, ocean and atmosphere models, for instance.~~

5.2 Physical Constants

`sympl` maintains a unit-aware library of constants which can be accessed or modified by model packages and by the user through `get_constant()` and `set_constant()` functions. For example, `planetary_rotation_rate` can be changed ~~to perform sensitivity experiments~~with a single function call. The unit handling is important to ensure constants are given to components in the units ~~required~~they each require. For example, the RRTMG radiative transfer code (Clough et al., 2005), requires physical constants in CGS units.

For the purposes of logging, physical constants are classified into various categories:

- ~~Planetary constants like~~ planetary constants such as rotation rate, acceleration due to gravity.
- physical constants ~~like~~ such as the speed of light.
- atmospheric constants ~~like~~ such as specific heat of dry air and reference air pressure.
- stellar constants ~~like~~ such as stellar irradiance

- condensible constants which refer to the thermodynamic properties of the condensible (in all three phases) in the atmosphere.
- Oceanographic constants such as the reference sea water density.

We chose to keep the constants related to the condensible component of the atmosphere separate to ensure `sympl` is flexible enough to handle general planetary atmospheres. `sympl` provides a function `set_condensible()` which allows switching all constants related to the condensible. For example `set_condensible('methane')` will replace all condensible constants (such as density of liquid/solid/gaseous phases, latent heat of condensation) to those corresponding to methane, provided such constants are already in the constants dictionary. The default condensible is water, which is currently the only condensible compound for which default values are given ~~.~~

~~-~~

5.3 ~~Anatomy of a `sympl` component~~

~~A by~~ `sympl`~~component is described in Fig. 4. The general code layout for a `sympl` component. `input_properties` and `tendency_properties` are examples of property attributes that contain details of the required inputs and returned outputs, tendencies or diagnostics. Here, the input is a quantity called '`air_temperature`' whose horizontal dimensions are `latitude`, `longitude` and whose values in the vertical are defined at model mid-levels. The units are specified for each quantity.~~

~~Dimension and unit requirements in `sympl` are not restrictions on the inputs, but rather describe the internal representation used by the component. `sympl` provides helper functions that will automatically convert the input state to satisfy these requirements, and raise an exception if that is not possible~~.

~~Since this component is a `Prognostic`, it outputs tendencies of a quantity called `air_pressure` whose dimensions are the same as that of the input quantity. The `__call__()` method accepts the state dictionary as input and returns tendencies as specified in the component properties. If the component was an `Implicit`, then the above method would also require the timestep as an argument to step the quantities forward in time.~~

5.3 Modelling using `sympl`

A typical workflow when using a model written using `sympl`, as seen in previous examples such as Fig. 2 and Fig. 3, might involve the following steps:

1. Initialise model components, providing configuration information.
2. Use `Wrapper` components to modify the behaviour of any components if necessary.
3. Intialise model state which contains all quantities required by the selected components.
4. Use ~~`TimeStepper`~~`TendencyStepper` to collect all ~~`Prognostic`~~`TendencyComponent` components into a component that can ~~be stepped~~ step the model state forward in time.

5. ~~Call~~ Begin the model main loop.

6. Call ~~Diagnostic~~DiagnosticComponent to compute any derived quantities from prognostic quantities or provide forcing quantities at a given time step.

7. Call ~~Implicit~~Stepper components and get a new state dictionary with the updated model quantities and any diagnostics. Update the initial model state with ~~new values and~~ diagnostics.

8. Call ~~TimeStepper~~TendencyStepper and get a new state dictionary with the updated model quantities and any diagnostics. Update ~~model state~~the initial model state with diagnostics.

9. Call any Monitor components to store ~~model state~~[4] the initial model state (e.g. store to disk, display in real time, send over the network).

10. ~~Increment model time.~~ Increment model time and repeat the model main loop.

~~A schematic version of the data flow in a model run is presented in Fig. 6. The TimeStepper provides the same model state to all Prognostics it contains and sums the tendencies before stepping forward in time. Using other time marching algorithms such as sequential tendency or sequential update splitting (Donahue and Caldwell, 2018) will require the user to implement their own TimeStepper subclass. All components are called in serial order, though there is no design restriction which forbids calling the components in parallel at each time step.~~

~~Flow of data for each type of component in a typical sympl+climt modelling run. The four panels show **a)** Diagnostic components **b)** Monitor components **c)** TimeStepper with Prognostic components **d)** Implicit components. The dark green boxes denote components, light green boxes denote optional Wrappers, the orange boxes indicate the state dictionary at different times and the purple boxes indicate tendencies and diagnosties generated by components.~~

6 `climt` – Design and programming interface

~~While sympl focuses on providing a programming model, a rich ontology of components and model agnostic configuration options,~~

6.1 Model state, quantity dimensions and output dictionaries

For initialisation, climt ~~focuses on adding the actual scientific components, model dependent configuration options and some helper functions to reduce boilerplate code while writing components.~~ provides the functions ~~climtalso adds additional attributes to existing components required to initialise model state, which is model dependent and not handled by~~ get_grid() and ~~sympl~~get_default_state().

[4] ~~Note that "store" could mean to store to disk, display in real time, send over the network, or anything else.~~

~~climt~~get_grid() ~~also introduces the ClimtSpectralDynamicalCore component which is a subclass of TimeStepper and is used to represent spectral dynamical cores. Spectral dynamical cores typically step the model forward in spectral space, and therefore require tendencies in spectral space.~~

6.2 ~~Model state, quantity dimensions and output dictionaries~~

~~climt provides a function~~ creates quantities that define the grid such as latitude, longitude, and air pressure. get_default_state() ~~which~~ accepts a list of components ~~(of any kind) and optional coordinate values for the four basic dimensions (defined as x, y, interface_levels and mid_levels) and~~ and optionally a state with grid quantities and creates a state dictionary ~~. The shape of the arrays in the resulting state dictionary is determined by the values of these coordinates. In addition to dimensions and units, climt also requires component properties to specify default values for~~ which satisfies the input requirements for those components. Default values of each model quantity ~~. A dictionary is maintained internally~~ are defined centrally in climt ~~for the most commonly used model quantities~~[4]. The default values ~~for these internally defined quantities~~ provided are scientifically meaningful, and can be used without modification for certain simulations.

~~Certain complications arise when creating the model state, and these are handled by climt in the following manner: Certain model quantities may have dimensions which do not correspond to any spatial dimension. For instance, correlated-k radiative transfer codes require arrays with an additional dimension that corresponds to the number of bands used to discretise the electromagnetic spectrum. climt allows components to specify these dimensions and their coordinate values by allowing an optional attribute called extra_dimensions. Certain model quantities may already have a description in the internal climt dictionary, but a component may want to describe it differently for algorithmic purposes. To allow for this use case, climt allows components to redefine the dimensions, units, and other attributes of a model quantity by defining an optional attribute quantity_descriptions.~~

~~For components with a large number of outputs, tendencies or diagnostics, creating the output dictionaries can consume many lines of code. climt components define a method create_state_dict_for() to reduce this boilerplate and keep component code readable.~~

6.2 Model Composition

Currently, the creation of the model and running the simulation loop is done by hand, which provides better understanding of what the model is doing but ~~in many cases is standardised and repetitive~~increases the verbosity of model code. In the near future, climt will provide an additional class called Federation which automates the process of creating a model from its components. Federation would not require that the user know the difference between a TendencyComponent and Stepper (for instance) and their different call signatures, or that TendencyComponents require a TendencyStepper to step the model state forward in time. This makes creating models easy especially for those who are not familiar with climate modelling. The tradeoff is that the run script will not explicitly describe the sequence of the main loop because that information

[4] ~~As of this writing, these quantities are mainly from the atmosphere and land domains.~~

is hidden within the `Federation` code, but this can be desirable for certain applications, particularly in education. As mentioned before, this automation is possible only because of the rich taxonomy of components `sympl` provides.

7 ~~Features and software engineering~~

6.1 Features and software engineering

`climt` currently has the following components that can be used to build models[4]:

- RRTMG longwave and shortwave radiative transfer (Clough et al., 2005): This is a Fortran component accessed via a cython wrapper. RRTMG is a state-of-the-art radiative transfer code used in many climate models.
- Grey gas radiation scheme: ~~along with a `Diagnostic` component~~ simulates radiative transfer in a grey gas. This component is accompanied by another component that provides an optical depth distribution which mimics the effect of water vapour (Frierson et al., 2006). These components are written in pure Python. This radiative scheme has been used in many idealised climate dynamics simulations to isolate the thermodynamic effects of latent heat release from the radiative effects of water vapour, which is a strong greenhouse gas.
- Insolation: This component is written in pure Python. It provides the solar zenith angle based on the time available in the model state. This zenith angle is used in radiative transfer codes. Currently, this component uses approximations and orbital parameters which make it highly accurate for earth but inapplicable to other planets.
- Emanuel convection scheme (Emanuel and Živković Rothman, 1999): This is a Fortran component accessed via a ~~cython~~ Cython wrapper. It is a mass flux based convection scheme which is based on the boundary layer quasi-equilibrium hypothesis (Raymond, 1995).
- Grid scale condensation~~,~~: This is written in pure Python. It calculates the water vapour and temperature fields in the atmosphere after condensing out excess water vapour to keep the atmospheric column from becoming super-saturated.
- Spectral dynamical core~~,~~: This component is derived from the General Forecast System (https://github.com/jswhit/gfs-dycore). ~~This~~ It is a Fortran module accessed via a cython wrapper. It uses a high performance spherical harmonics library `shtns` (https://bitbucket.org/nschaeff/shtns). It is parallelised using OpenMP (Dagum and Menon, 1998), and therefore is most effective on shared memory systems. The dynamics is stepped using an implicit-explicit total variation diminishing Runge-Kutta 3 time-stepper. The physics tendencies are stepped forward using a forward Euler scheme.
- Simple Physics package for idealised simulations (Reed and Jablonowski, 2012)~~.~~: This is a fortran module accessed via a cython wrapper. It provides initial conditions which can be used for testing moist dynamical cores, and also provides a simple diffusive boundary layer suitable for idealised simulations.

[4] All components written in pure Python were written by the authors for use in `climt`.

- Slab surface~~.~~: This component is written in pure Python. It allows for a prognostic surface temperature by calculating the surface energy budget. It is flexible enough to represent land or ocean. It currently does not account for localised heat fluxes.

- Sea/land ice model which allows for snow and ice layers, and energy balanced top and bottom surfaces. This component is written in pure Python. It is flexible enough to represent ice/snow growth and melting. It is capable of representing sea or land ice based on the surface type available in the model state. It currently cannot handle fractional land surface types.

- Held-Suarez forcing (Held and Suarez, 1994)~~.~~: This component is written in pure Python. It provides an idealised set of model physics which can be used for testing dry dynamical cores and idealised simulations.

- Initial conditions from the dynamical core MIP (DCMIP) ~~.~~(https://www.earthsystemcog.org/projects/dcmip/): This is a Fortran module accessed via a cython wrapper. It provides initial conditions for a wide variety of tests which allow assessing the conservation properties of dynamical cores.

This set of components allow building a hierarchy of models ranging from single column radiative-convective models to energy balanced moist atmospheric general circulation models. Because of the fine-grained configurability of `sympl`/`climt`, the difference between the number of lines of code required to build a single column model and a moist GCM is only around ~~40~~ 10 lines of Python code ~~;~~(see scripts in supplementary material). More importantly, most of the code is reusable when moving from a simpler to a more complex model.

Both `sympl` and `climt` are open source projects, licensed under a permissive BSD license. Both packages are available on Mac, Linux and Windows platforms, and can be directly installed from the Python Package Index using one line commands:

```
pip install sympl
pip install climt
```

eliminating the need to download source code from GitHub. The Python Package Index projects are located at https://pypi.python.org/pypi/sympl and https://pypi.python.org/pypi/climt respectively.

`climt` also provides binary releases on all supported platforms, eliminating the need to have a compiler on the user's system. `sympl` is written in pure Python, and does not have any compiler requirements. Both packages are regression tested using the online services TravisCI (https://travis-ci.org/) and AppVeyor (https://www.appveyor.com/). Both packages also maintain regularly updated documentation at http://sympl.readthedocs.io/en/latest/ and http://climt.readthedocs.io/en/latest/.

7 ~~Example Script~~

~~Figure ?? shows a typical script used to build a model using `sympl` and `climt`. The first few lines simply import components that are required to build the model. These imports also serve to inform the user which components are required to build the required model.~~

~~Lines 8-20 initialise the components and the model timestep. This includes a variety of components, including `Monitor`, `Prognostic` and `Implicit` components. Algorithmic configuration information is passed to the components as keyword arguments while initialising them.~~

~~Lines 22-31 are concerned with creating a model state and initialising suitable values. This initialisation is dependent on the scientific question at hand.~~

~~A TimeStepper is created on Line 33 which combines the tendencies from the various `Prognostics` and will eventually produce new values of state quantities.~~

~~Lines 37-49 are concerned with running the model itself. The state updated by the `TimeStepper` is used as input to the `simple_physics` component which generates a new state. The diagnostics generated during these calculations are collected in the old state variable which can then be stored. Finally, the state variable is updated and the model timestep is incremented.~~

~~Changing the simulation to something other than a single-column model requires that information about the spatial dimensions be passed to the `get_default_state` function. Therefore, converting this script to a GCM will only require initialising the dynamical core component (which acts as the time stepper) and specifying the appropriate dimensional information while creating model state. Example script for a single-column radiative-convective model. Some code was omitted to enhance the presentation.~~

7 Some benchmark simulations

The first simulation is that of an atmospheric column that is run to equilibrium in the presence of radiation and convection. This model uses the RRTMG longwave and shortwave components, the Emanuel convection scheme, the Simple Physics component as its boundary layer scheme and a slab ocean of thickness 50 ~~meters at the surface~~metres. The model timestep is 5 minutes and the results presented in Fig. 8 are the mean between ~~8000 and 10000~~ 15000 and 20000 timesteps. The ~~results are presented in Fig. 8. The~~ air temperature transitions from a dry adiabat in the boundary layer to a moist adiabat in the free atmosphere until the tropopause. ~~Correspondingly, the equivalent potential temperature increases slowly until the tropopause and rapidly thereafter~~The diabatic heating balance changes from a balance between radiation and convection in the troposphere of the model to a pure radiative equilibrium in the stratosphere.

The second simulation is of a idealised aquaplanet GCM with fixed equinoctial insolation. As mentioned before, the modular nature of our framework allows re-use of much of the runscript code from the above single column model. It consists of all the components used in the previous model along with a dynamical core which is used as the time stepper. The model was run for two years and the results presented are the mean over the last six months. The simulated climate of the model is as expected from such a configuration: the zonal mean zonal winds show two strong westerly jets which penetrate to the surface. The zonal mean temperature shows a distinct tropical cold point and an increase in the temperature above the tropopause. The zonal mean convective heating rate shows deep heating in the tropics and much shallower heating in the subtropical areas dominated by descent of air. This simulation ran at a resolution of 128 longitudes, 62 latitudes (or T42 resolution) and 28 levels. ~~It was run using 16 cores (Intel Xeon, 3.10 GHz), and the throughput was approximately 7 hours per model year. The server on which it~~

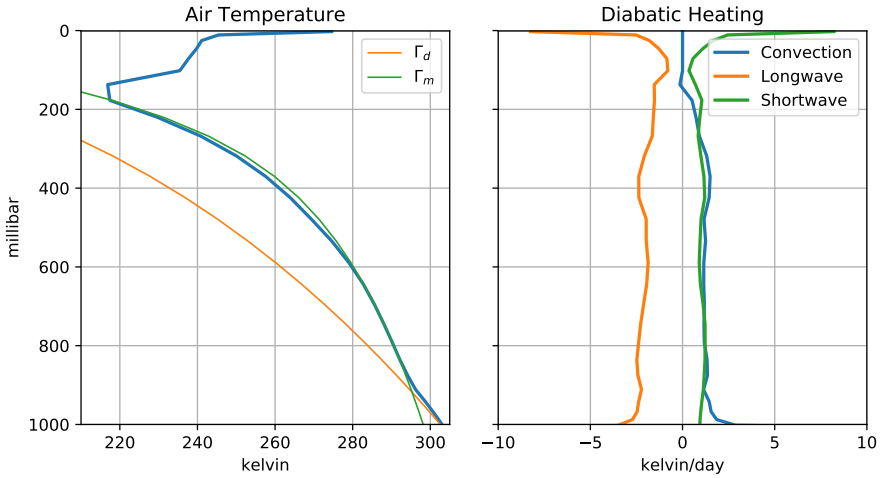

Figure 8. The mean equilibrium profiles in the radiative convective single column model. ~~In~~ The mean temperature is presented in the ~~top~~ left panel, along with the dry and moist adiabats which are ~~plotted in red~~ orange and green respectively. The mean heating profiles are presented in the right panel, which shows a stratosphere in radiative equilibrium and a troposphere in radiative-convective equilibrium.

~~was run was a shared server which means that these performance figures should probably be a lower bound. We also expect the performance to improve further since currently only the dynamical core and the radiative transfer components run in parallel (using openMP).~~

8 Conclusions and Future Avenues

`sympl` and `climt` represent a novel approach to climate modelling which provides the user with fine-grained control over the configuration of the model. `sympl` provides a rich set of entities which describe all functionality typically expected of a climate model. This set of entities (or classes) allows `climt` to be an easy to use climate modelling toolkit by allowing decisions about model creation and configuration to be made at a single location (the run script) and without ambiguity. The modular nature of the packages allows for code reuse as one traverses the hierarchy of models from single column model to three dimensional GCMs. We attempt to address concerns about plug-and-play type architectures (Randall, 1996) by ensuring the inputs and outputs of each model are cleanly documented, which makes it clear whether components are compatible or not. The use of Python allows for delegating computationally intensive code to compiled languages while still providing an intuitive and clean interface to the user. This choice also allows users access to a large variety of libraries written in Python for purposes ranging from machine learning to visualisation (Alpire, 2017).

The main focus in the near future would be to add more components, especially a cloud microphysics scheme, to allow `sympl`/`climt` to simulate a more realistic benchmark climate~~, especially a cloud microphysics scheme~~. Due to its flexibility, we believe our modelling framework is well suited to the simulation of ~~planetary atmosphere~~ general planetary atmospheres

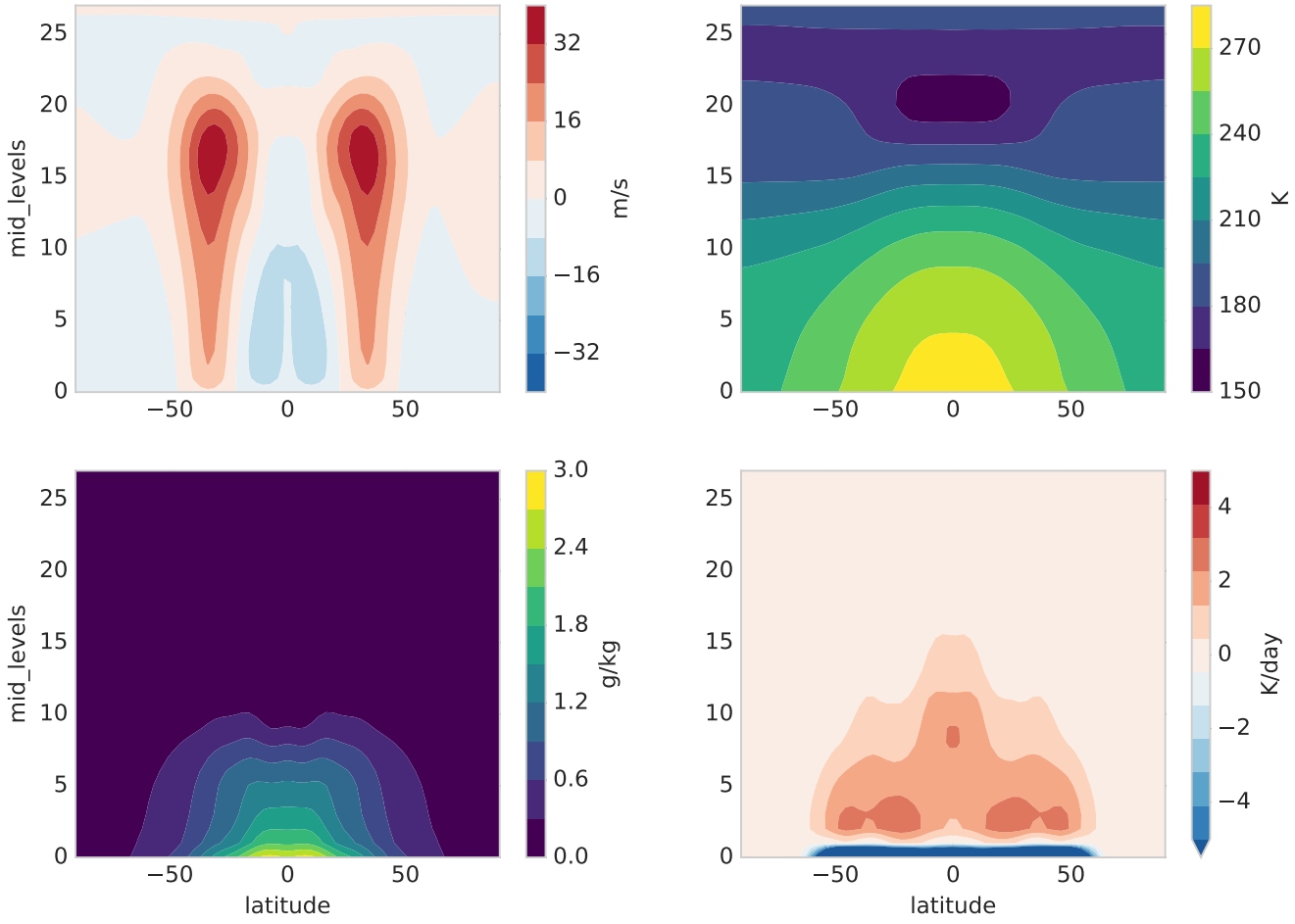

Figure 9. The zonal mean equilibrium profiles in the idealised GCM runs with no seasonal cycle. The plotted fields are, in clockwise order from the top left, the zonal winds, air temperature, convective heating rate and specific humidity respectively. The y-axis is in model levels and x axis is in degrees.

and for exoplanet ~~research, and efforts towards~~ modelling, and adding components relevant to these fields will also be a priority. Another important component to add would be a flexible grid interpolation component to allow interaction between components based on different model grids. ~~Modifications are also being made to `sympl` to further simplify the writing of model components, with automatic validation of input and output properties and automatic preparation of `numpy` arrays for use by components.~~

While care has been taken to ensure that parallel computing is possible, we have yet to address the question of distributed memory and computing. While ~~running our model~~ building models in a simple MPI scenario seems feasible in the near future, more sophisticated configurations with components running in parallel will need some thought and design.

Nevertheless, `sympl` and `climt` represent an important step towards creating flexible, usable and readable models. We hope that they will be a useful addition to the growing collection of Python based tools available to the climate science community.

Code availability. `sympl` is available at https://github.com/mcgibbon/sympl. The digital object identifier (DOI) for the version documented in this paper is https://zenodo.org/record/1346405.
`climt` is available at https://github.com/CliMT/climt. The digital object identifier (DOI) for the version documented in this paper is https://zenodo.org/record/1400103.

Acknowledgements. The first and third authors acknowledge a research grant from the Swedish e-Science Research Center (SeRC). The second author was funded by DOE grant DE-SC0016433 as a contribution to the CMDV (CM)4 project, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship Doctoral Program (PGS-D).

References

Alpire, A.: Predicting Solar Radiation using a Deep Neural Network, http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-215715, 2017.

Balaji, V., Benson, R., Wyman, B., and Held, I.: Coarse-grained component concurrency in Earth system modeling: parallelizing atmospheric radiative transfer in the GFDL AM3 model using the Flexible Modeling System coupling framework, Geosci. Model Dev., 9, 3605–3616, https://doi.org/10.5194/gmd-9-3605-2016, https://www.geosci-model-dev.net/9/3605/2016/, 2016.

Clough, S. A., Shephard, M. W., Mlawer, E. J., Delamere, J. S., Iacono, M. J., Cady-Pereira, K., Boukabara, S., and Brown, P. D.: Atmospheric radiative transfer modeling: a summary of the AER codes, Journal of Quantitative Spectroscopy and Radiative Transfer, 91, 233–244, https://doi.org/10.1016/j.jqsrt.2004.05.058, http://www.sciencedirect.com/science/article/pii/S0022407304002158, 2005.

Dagum, L. and Menon, R.: OpenMP: an industry standard API for shared-memory programming, IEEE computational science and engineering, 5, 46–55, 1998.

DeLuca, C., Theurich, G., and Balaji, V.: The Earth System Modeling Framework, in: Earth System Modelling - Volume 3, SpringerBriefs in Earth System Sciences, pp. 43–54, Springer, Berlin, Heidelberg, https://link-springer-com.ezp.sub.su.se/chapter/10.1007/978-3-642-23360-9_6, dOI: 10.1007/978-3-642-23360-9_6, 2012.

Donahue, A. S. and Caldwell, P. M.: Impact of Physics Parameterization Ordering in A Global Atmosphere Model, Journal of Advances in Modeling Earth Systems, pp. n/a–n/a, https://doi.org/10.1002/2017MS001067, http://onlinelibrary.wiley.com/doi/10.1002/2017MS001067/abstract, 2018.

Emanuel, K. A. and Živković Rothman, M.: Development and Evaluation of a Convection Scheme for Use in Climate Models, Journal of the Atmospheric Sciences, 56, 1766–1782, https://doi.org/10.1175/1520-0469(1999)056<1766:DAEOAC>2.0.CO;2, http://journals.ametsoc.org.ezp.sub.su.se/doi/abs/10.1175/1520-0469(1999)056%3C1766%3ADAEOAC%3E2.0.CO%3B2, 00553, 1999.

Fraedrich, K., Jansen, H., Kirk, E., Luksch, U., and Lunkeit, F.: The Planet Simulator: Towards a user friendly model, Meteorologische Zeitschrift, 14, 299–304, https://doi.org/10.1127/0941-2948/2005/0043, 2005.

Frierson, D. M. W., Held, I. M., and Zurita-Gotor, P.: A Gray-Radiation Aquaplanet Moist GCM. Part I: Static Stability and Eddy Scale, Journal of the Atmospheric Sciences, 63, 2548–2566, https://doi.org/10.1175/JAS3753.1, http://journals.ametsoc.org/doi/abs/10.1175/JAS3753.1, 2006.

Held, I. M.: The Gap between Simulation and Understanding in Climate Modeling, Bulletin of the American Meteorological Society, 86, 1609–1614, https://doi.org/10.1175/BAMS-86-11-1609, http://journals.ametsoc.org/doi/abs/10.1175/BAMS-86-11-1609, 2005.

Held, I. M. and Suarez, M. J.: A Proposal for the Intercomparison of the Dynamical Cores of Atmospheric General Circulation Models, Bulletin of the American Meteorological Society, 75, 1825–1830, https://doi.org/10.1175/1520-0477(1994)075<1825:APFTIO>2.0.CO;2, http://journals.ametsoc.org/doi/abs/10.1175/1520-0477%281994%29075%3C1825%3AAPFTIO%3E2.0.CO%3B2, 1994.

Hoyer, S. and Hamman, J. J.: xarray: N-D labeled Arrays and Datasets in Python, Journal of Open Research Software, 5, https://doi.org/10.5334/jors.148, https://doi.org/10.5334/jors.148, 2017.

Jeevanjee, N., Hassanzadeh, P., Hill, S., and Sheshadri, A.: A perspective on climate model hierarchies, Journal of Advances in Modeling Earth Systems, pp. n/a–n/a, https://doi.org/10.1002/2017MS001038, http://onlinelibrary.wiley.com.ezp.sub.su.se/doi/10.1002/2017MS001038/abstract, 2017.

Peng, R. D.: Reproducible Research in Computational Science, Science, 334, 1226–1227, https://doi.org/10.1126/science.1213847, http://science.sciencemag.org.ezp.sub.su.se/content/334/6060/1226, 2011.

Randall, D. A.: A University Perspective on Global Climate Modeling, Bulletin of the American Meteorological Society, 77, 2685–2690, https://doi.org/10.1175/1520-0477(1996)077<2685:AUPOGC>2.0.CO;2, https://journals.ametsoc.org/doi/abs/10.1175/1520-0477%281996%29077%3C2685%3AAUPOGC%3E2.0.CO%3B2, 1996.

Raymond, D. J.: Regulation of Moist Convection over the West Pacific Warm Pool, Journal of the Atmospheric Sciences, 52, 3945–3959, https://doi.org/10.1175/1520-0469(1995)052<3945:ROMCOT>2.0.CO;2, https://journals.ametsoc.org/doi/abs/10.1175/1520-0469%281995%29052%3C3945%3AROMCOT%3E2.0.CO%3B2, 1995.

Reed, K. A. and Jablonowski, C.: Idealized tropical cyclone simulations of intermediate complexity: a test case for AGCMs, Journal of Advances in Modeling Earth Systems, 4, http://onlinelibrary.wiley.com/doi/10.1029/2011MS000099/full, 2012.

Theurich, G., DeLuca, C., Campbell, T., Liu, F., Saint, K., Vertenstein, M., Chen, J., Oehmke, R., Doyle, J., Whitcomb, T., Wallcraft, A., Iredell, M., Black, T., Da Silva, A. M., Clune, T., Ferraro, R., Li, P., Kelley, M., Aleinov, I., Balaji, V., Zadeh, N., Jacob, R., Kirtman, B., Giraldo, F., McCarren, D., Sandgathe, S., Peckham, S., and Dunlap, R.: The Earth System Prediction Suite: Toward a Coordinated U.S. Modeling Capability, Bulletin of the American Meteorological Society, 97, 1229–1247, https://doi.org/10.1175/BAMS-D-14-00164.1, https://journals.ametsoc.org/doi/abs/10.1175/BAMS-D-14-00164.1, 2015.

Valcke, S., Redler, R., and Budich, R.: Earth System Modelling - Volume 3, SpringerBriefs in Earth System Sciences, Springer Berlin Heidelberg, Berlin, Heidelberg, http://link.springer.com/10.1007/978-3-642-23360-9, dOI: 10.1007/978-3-642-23360-9, 2012.

Vallis, G. K., Colyer, G., Geen, R., Gerber, E., Jucker, M., Maher, P., Paterson, A., Pietschnig, M., Penn, J., and Thomson, S. I.: Isca, v1.0: A Framework for the Global Modelling of the Atmospheres of Earth and Other Planets at Varying Levels of Complexity, Geosci. Model Dev. Discuss., 2017, 1–25, https://doi.org/10.5194/gmd-2017-243, https://www.geosci-model-dev-discuss.net/gmd-2017-243/, 2017.