1 Installation 3

2 Usage 5
  2.1 Basic demos ................................................................. 5
  2.2 More detail ................................................................. 27

3 Development 47
  3.1 Contributing ............................................................... 47
  3.2 Getting setup ............................................................. 48
  3.3 Formatting ................................................................. 49
  3.4 Building the docs ....................................................... 49
  3.5 Releasing ................................................................. 50
  3.6 Why is there a makefile in a pure Python repository? ........... 50

4 Base API 51

5 Impulse Response Model API 53

6 Two Layer Model API 57

7 Constants API 61

8 Errors API 63

9 Utils API 65

10 Changelog 67
  10.1 master ................................................................. 67
  10.2 v0.2.0 - 2020-10-09 ............................................... 67
  10.3 v0.1.2 - 2020-31-07 ............................................... 68
  10.4 v0.1.1 - 2020-06-29 ............................................... 68
  10.5 v0.1.0 - 2020-05-15 ............................................... 68

11 Index 69

Python Module Index 71

Index 73
OpenSCM two layer model contains implementations of the two layer radiative forcing driven models by Held et al., Geoffroy et al. and Bloch-Johnson et al.

OpenSCM two layer model is free software under a BSD 3-Clause License, see LICENSE.
INSTALLATION

OpenSCM two layer model can be installed with pip

```
pip install openscm-twolayermodel
```

If you also want to run the example notebooks install additional dependencies using

```
pip install "openscm-twolayermodel[notebooks]"
```

**Coming soon** OpenSCM two layer model can also be installed with conda

```
conda install -c conda-forge openscm-twolayermodel
```
Chapter 1. Installation
Here we provide examples of OpenSCM two layer model’s behaviour and usage. The source code of these usage examples is available in the folder `docs/source/usage` of the GitHub repository.

### 2.1 Basic demos

#### 2.1.1 Getting Started

This notebook demonstrates the OpenSCM Two Layer Model repository’s basic functionality.

We start with imports, their need will become clearer throughout the notebook.

```
[1]: import inspect

import numpy as np
from openscm_units import unit_registry
from scmdata import ScmRun

import openscm_twolayermodel
from openscm_twolayermodel import ImpulseResponseModel, TwoLayerModel
from openscm_twolayermodel.base import Model

/home/docs/checkouts/readthedocs.org/user_builds/openscm-two-layer-model/envs/17-nbsphinx/lib/python3.7/site-packages/openscm_twolayermodel/base.py:10:
    TqdmExperimentalWarning: Using `tqdm.autonotebook.tqdm` in notebook mode. Use `tqdm.tqdm` instead to force console mode (e.g. in jupyter console)

import tqdm.autonotebook as tqdm

As with most Python packages, the version of `openscm_twolayermodel` being used can always be checked as shown below. This is very helpful for debugging.

```
[2]: # NBVAL_IGNORE_OUTPUT

openscm_twolayermodel.__version__

```

```
[2]: '0.2.0+25.ge87e2fb'

```

OpenSCM Two Layer Model has two key classes: `ImpulseResponseModel` and `TwoLayerModel`. These are implementations of the two major variants of the two-layer model found in the literature. We can see that they both have a common base class using the `inspect` package.

```
[3]: inspect.getmro(ImpulseResponseModel)

```
These classes can both be used in the same way. We demonstrate the most basic usage here, more comprehensive usage is demonstrated in other notebooks.

The first thing we need is our effective radiative forcing driver. This should be an `ScmRun` instance.

```python
run_length = 200

driver = ScmRun(
    data=np.arange(run_length) * 4 / 70,
    index=1850 + np.arange(run_length),
    columns={
        "unit": "W/m^2",
        "model": "idealised",
        "scenario": "1pctCO2",
        "region": "World",
        "variable": "Effective Radiative Forcing",
    },
)
```

```python
driver.lineplot()
```

```text
<scmdata.ScmRun (timeseries: 1, timepoints: 200)>
```

Time:
  Start: 1850-01-01T00:00:00
  End: 2049-01-01T00:00:00

Meta:
          model region scenario      unit variable
0      idealised World 1pctCO2 W/m^2 Effective Radiative Forcing
```
Then we can initialise instances of our models and run them.

```python
# NBVAL_IGNORE_OUTPUT
two_layer = TwoLayerModel(lambda0=4 / 3 * unit_registry("W/m^2/delta_degC"))
res_two_layer = two_layer.run_scenarios(driver)

impulse_response = ImpulseResponseModel(d1=10 * unit_registry("yr"))
res_impulse_response = impulse_response.run_scenarios(driver)

res = res_two_layer.append(res_impulse_response)
res.head()
```

(continues on next page)
W/m^2 Heat Uptake 0.0
NaN two_timescale_impulse_response 10.0
400.0 NaN NaN 1.0 NaN
NaN NaN idealised 0.3
World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.0

time

1851-01-01 \(\frac{\text{a \ (watt / delta_degree_Celsius \times meter \times meter)}}{\text{a \ (delta_degree_Celsius \times meter \times meter) \ \text{climate_model \ \text{dl}}}}\)
\(\frac{\text{a \ (delta_degree_Celsius \times meter \times meter) \ \text{efficacy \ (dimensionless) \ \text{eta \ (watt / delta_degree_Celsius) \ \text{lambda0 \ (watt / delta_degree_Celsius / meter \times meter) \ \text{model}}}}}{\text{f \ (delta_degree_Celsius \times meter \times meter / watt) \ q2 \ (delta_degree_Celsius \times meter \times meter / watt) \ \text{region \ run_idx \ scenario \ unit \ variable}}\)
0.0 two_layer NaN
NaN 1200.0 50.0 1.0 0.8
1.333333 NaN idealised NaN
World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.057143

delta_degC Surface Temperature|Upper 0.000000
Surface Temperature|Lower 0.000000

W/m^2 Heat Uptake 0.000000
NaN two_timescale_impulse_response 10.0
400.0 NaN NaN 1.0 NaN
NaN NaN idealised 0.3
World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.057143

time

1852-01-01 \(\frac{\text{a \ (watt / delta_degree_Celsius \times meter \times meter)}}{\text{a \ (delta_degree_Celsius \times meter \times meter) \ \text{climate_model \ \text{dl}}}}\)
\(\frac{\text{a \ (delta_degree_Celsius \times meter \times meter) \ \text{efficacy \ (dimensionless) \ \text{eta \ (watt / delta_degree_Celsius) \ \text{lambda0 \ (watt / delta_degree_Celsius / meter \times meter) \ \text{model}}}}}{\text{f \ (delta_degree_Celsius \times meter \times meter / watt) \ q2 \ (delta_degree_Celsius \times meter \times meter / watt) \ \text{region \ run_idx \ scenario \ unit \ variable}}\)
2.1. Basic demos
### Chapter 2. Usage

<table>
<thead>
<tr>
<th>time</th>
<th>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1854-01-01</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>(a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_</td>
</tr>
<tr>
<td></td>
<td>–Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model</td>
</tr>
<tr>
<td>q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2)</td>
<td></td>
</tr>
<tr>
<td>/ watt)</td>
<td>region run_idx scenario unit variable</td>
</tr>
<tr>
<td>0.0</td>
<td>two_layer NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>NaN</td>
<td>World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.228571</td>
</tr>
<tr>
<td>NaN</td>
<td>delta_degC Surface Temperature</td>
</tr>
<tr>
<td>NaN</td>
<td>Surface Temperature</td>
</tr>
<tr>
<td>NaN</td>
<td>W/m^2 Heat Uptake NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>two_timescale_impulse_response 10.0</td>
</tr>
<tr>
<td>400.0</td>
<td>NaN NaN 1.0 NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.228571</td>
</tr>
<tr>
<td>time</td>
<td>1855-01-01</td>
</tr>
<tr>
<td>NaN</td>
<td>(a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_</td>
</tr>
<tr>
<td></td>
<td>–Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model</td>
</tr>
<tr>
<td>q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2)</td>
<td></td>
</tr>
<tr>
<td>/ watt)</td>
<td>region run_idx scenario unit variable</td>
</tr>
<tr>
<td>0.0</td>
<td>two_layer NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN 1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>NaN</td>
<td>World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.285714</td>
</tr>
<tr>
<td>NaN</td>
<td>delta_degC Surface Temperature</td>
</tr>
</tbody>
</table>

(continues on next page)
Surface Temperature|Lower 0.000368

W/m² Heat Uptake 0.173178 NaN
two_timescale_impulse_response 10.0
400.0 NaN NaN 1.0 NaN
NaN idealised 0.3
0.4
World 0 lpc	CO2 W/m² Effective Radiative Forcing 0.285714

time

1856-01-01 a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model d1
(a) d2 (a) dl (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_-
Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model
q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 /
watt) region run_idx scenario unit variable
0.0 two_layer NaN
NaN 1200.0 50.0 1.0 0.8 NaN
NaN idealised NaN
World 0 lpc	CO2 W/m² Effective Radiative Forcing 0.342857

delta_degC Surface Temperature|Upper 0.085676

Surface Temperature|Lower 0.000681

W/m² Heat Uptake 0.202129 NaN
two_timescale_impulse_response 10.0
400.0 NaN NaN 1.0 NaN
NaN idealised 0.3
0.4
World 0 lpc	CO2 W/m² Effective Radiative Forcing 0.342857

time

1857-01-01 (continues on next page)
We are modeling a system that has two layers, and we have two different models for different regions.

### Model 1: Two Layer Model

- **Variable**: `two_layer`
- **Unit**: `NaN`
- **Value**: `0.0`

### Model 2: Idealized Model

- **Variable**: `idealised`
- **Unit**: `NaN`
- **Value**: `1.333333`

### Effective Radiative Forcing

- **Region**: `World`
- **Scenario**: `1pctCO2`
- **Value**: `0.3`

### Surface Temperature

- **Upper Layer**: `0.109924`
- **Lower Layer**: `0.001109`

### Heat Uptake

- **Value**: `0.228623`

### Time Series

- **Start Date**: `1858-01-01`

### Additional Information

- **Other Models**: `a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl, a (a) d2 (a) dl (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / watt) region run_idx scenario unit variable 0.0 two_layer NaN NaN 1200.0 50.0 1.0 0.8 1.333333 NaN NaN World 0 1pctCO2 W/m^2 Effective Radiative Forcing 0.400000`
### 2.1. Basic demos

```
| NaN | two_timescale_impulse_response 10.0 |
| 400.0 | NaN | NaN | 1.0 | NaN |
| NaN | 0.4 |
| World 0 | lpctCO2 | W/m² | Effective Radiative Forcing | 0.457143 |

```

```
time

| 1859-01-01 | NaN | NaN | 1200.0 | 50.0 | 1.0 | 0.8 |
| NaN | 1.333333 |
| World 0 | lpctCO2 | W/m² | Effective Radiative Forcing | 0.514286 |

```

```
delta_degC Surface Temperature|Upper 0.160761

```

```
Surface Temperature|Lower 0.002328

```

```
W/m² Heat Uptake 0.277089

```

```
| NaN | two_timescale_impulse_response 10.0 |
| 400.0 | NaN | NaN | 1.0 | NaN |
| NaN | 0.4 |
| World 0 | lpctCO2 | W/m² | Effective Radiative Forcing | 0.514286 |

```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
\begin{verbatim}
  delta_degC  Surface Temperature|Upper  ...
  
  Surface Temperature|Lower  ...
  
  W/m^2  Heat Uptake  ...
  NaN  two_timescale_impulse_response 10.0
  400.0  NaN  NaN  1.0  NaN
  NaN  idealised 0.3
  0.4
  World 0 1pctCO2 W/m^2 Effective Radiative Forcing ...

  time

  2040-01-01 \a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model d1
  (a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / watt) region run_idx scenario unit variable
  0.0  two_layer NaN
  NaN  1200.0  50.0  1.0  0.8
  1.333333 NaN idealised NaN
  NaN  World 0 1pctCO2 W/m^2 Effective Radiative Forcing 10.857143

  
  delta_degC  Surface Temperature|Upper  5.710809
  
  Surface Temperature|Lower  1.937427
  
  W/m^2  Heat Uptake  3.230641
  NaN  two_timescale_impulse_response 10.0
  400.0  NaN  NaN  1.0  NaN
  NaN  idealised 0.3
  0.4
  World 0 1pctCO2 W/m^2 Effective Radiative Forcing 10.857143
\end{verbatim}
(continued from previous page)

time

2041-01-01

a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl,
  (a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_
  _Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model
  q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / 
  watt) region run_idx scenario unit variable
0.0 two_layer NaN
  NaN 1200.0 50.0 1.0 0.8
  1.333333 idealised NaN
  NaN

World 0 lpctCO2 W/m^2 Effective Radiative Forcing 10.914286

delta_degC Surface Temperature|Upper 5.744627

Surface Temperature|Lower 1.956414

W/m^2 Heat Uptake 3.242731
NaN two_timescale_impulse_response 10.0
  400.0 NaN NaN 1.0 NaN
  NaN idealised 0.3
  0.4

World 0 lpctCO2 W/m^2 Effective Radiative Forcing 10.914286

time

2042-01-01

a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl,
  (a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_
  _Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model
  q1 (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / 
  watt) region run_idx scenario unit variable
0.0 two_layer NaN
  NaN 1200.0 50.0 1.0 0.8
  1.333333 idealised NaN
  NaN

World 0 lpctCO2 W/m^2 Effective Radiative Forcing 10.971429

delta_degC Surface Temperature|Upper 5.778474

(continues on next page)
Continued from previous page:

```
  W/m² Heat Uptake 3.254783
NaN two_timescale_impulse_response 10.0
  400.0 NaN NaN 1.0 NaN
  NaN NaN 0.4
  World 0 1pctCO2 W/m² Effective Radiative Forcing 10.971429
time
  
  2043-01-01
```

```
a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl
  (a) d2 (a) dl (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model
  q (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / watt) region run_idx scenario unit variable
```

```
0.0 two_layer NaN
  NaN 1200.0 50.0 1.0 0.8
  1.333333 NaN
  World 0 1pctCO2 W/m² Effective Radiative Forcing 11.028571
  
  delta_degC Surface Temperature|Upper 5.812348
  
  Surface Temperature|Lower 1.994612
  
  W/m² Heat Uptake 3.266797
NaN two_timescale_impulse_response 10.0
  400.0 NaN NaN 1.0 NaN
  NaN NaN 0.4
  World 0 1pctCO2 W/m² Effective Radiative Forcing 11.028571
time
  
  2044-01-01
```

```
a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl
  (a) d2 (a) dl (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model
  q (delta_degree_Celsius * meter ** 2 / watt) q2 (delta_degree_Celsius * meter ** 2 / watt) region run_idx scenario unit variable
```

(continues on next page)
<table>
<thead>
<tr>
<th>Time</th>
<th>Region</th>
<th>Scenario</th>
<th>W/m² Effective Radiative Forcing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>World</td>
<td>1pctCO₂</td>
<td>11.085714</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Surface Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Surface Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W/m² Heat Uptake</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>two_timescale_impulse_response</td>
</tr>
<tr>
<td>0.0</td>
<td>World</td>
<td>1pctCO₂</td>
<td>11.142857</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Surface Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Surface Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W/m² Heat Uptake</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>two_timescale_impulse_response</td>
</tr>
</tbody>
</table>

2.1. Basic demos
<table>
<thead>
<tr>
<th>time</th>
<th>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2046-01-01</td>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
</tr>
<tr>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
<td></td>
</tr>
<tr>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
<td></td>
</tr>
<tr>
<td>(a) d2 (a) d1 (meter) du (meter) efficacy (dimensionless) eta (watt / delta_degree_Celsius / meter ** 2) lambda0 (watt / delta_degree_Celsius / meter ** 2) model</td>
<td></td>
</tr>
<tr>
<td>2046-01-01</td>
<td>1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>2047-01-01</td>
<td>1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>2046-01-01</td>
<td>1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>2047-01-01</td>
<td>1200.0 50.0 1.0 0.8</td>
</tr>
<tr>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
<td></td>
</tr>
<tr>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
<td></td>
</tr>
<tr>
<td>a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model dl</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>2046-01-01</td>
<td>NaN</td>
</tr>
<tr>
<td>2047-01-01</td>
<td>NaN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>region</th>
<th>run_idx</th>
<th>scenario</th>
<th>unit</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>0</td>
<td>1pctCO2</td>
<td>W/m^2</td>
<td>Effective Radiative Forcing 11.200000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>delta_degC Surface Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Surface Temperature</td>
</tr>
<tr>
<td>W/m^2 Heat Uptake</td>
<td>3.302615</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>World</td>
<td>0</td>
<td>1pctCO2</td>
<td>W/m^2</td>
<td>Effective Radiative Forcing 11.257143</td>
</tr>
</tbody>
</table>

(continues on next page)
(continued from previous page)

    | Surface Temperature|Lower | 2.071897 |
    | W/m^2 Heat Uptake | 3.314480 |

NaN two_timescale_impulse_response 10.0

| 400.0 NaN NaN 1.0 NaN |
| NaN NaN | idealised 0.3 |
| World 0 1pctCO2 W/m^2 Effective Radiative Forcing 11.257143 |

2.1. Basic demos
Now we can plot our outputs and compare (of course, we can make these two models the same if we’re clever about how we set the parameters, see the impulse response equivalence notebook).

```
# NBVAL_IGNORE_OUTPUT
res.filter(variable="Surface Temperature*").lineplot(
    hue="climate_model", style="variable"
)
```

```
[8]: <AxesSubplot:xlabel='time', ylabel='delta_degC'>
```
2.1.2 Running scenarios

Here we show how multiple scenarios can be run using the OpenSCM Two Layer Model package.

```python
# NBVAL_IGNORE_OUTPUT
import os.path
import numpy as np
import pandas as pd
import openscm_units.unit_registry as ur
import tqdm.autonotebook as tqdman
from scmdata import ScmRun, run_append
```

(continues on next page)
from openscm_twolayermodel import TwoLayerModel

import matplotlib.pyplot as plt

DATA_PATH = os.path.join("..", "..", "..", "tests", "test-data", "rcmip-radiative-forcing-annual-means-v4-0-0.csv")

# NBVAL_IGNORE_OUTPUT
scenarios = ScmRun(DATA_PATH, lowercase_cols=True).filter(scenario="historical", keep=False)

scenarios

For this we use RCMIP effective radiative forcing data.
We can then run them, for a number of parameter settings, as shown.

```
# NBVAL_IGNORE_OUTPUT
a_values = np.array([0, 0.01]) * ur("W/m^2/delta_degC^2")
a_values
```

```
(0.0 0.01) watt
(delta_degree_Celsius* meter*)
```

```
# NBVAL_IGNORE_OUTPUT
runner = TwoLayerModel()
output = []
for a in tqdm(a_values, desc="Parameter settings"):
    runner.a = a
    output.append(runner.run_scenarios(scenarios))
output = run_append(output)
output
```

```
HBox(children=(HTML(value='Parameter settings'), FloatProgress(value=0.0, max=2.0),
              HTML(value='')))
```

```
HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
              layout=Layout(width='20px')...)
```

```
<scmdata.ScmRun (timeseries: 80, timepoints: 751)>
Time:
Start: 1750-01-01T00:00:00
End: 2500-01-01T00:00:00
Meta:
   a (watt / delta_degree_Celsius ** 2 / meter ** 2) activity_id
  0 0.00 not_applicable
  1 0.00 not_applicable
  2 0.00 not_applicable
  3 0.00 not_applicable
  4 0.00 not_applicable
.. ... ...
75 0.01 not_applicable
76 0.01 not_applicable
77 0.01 not_applicable
78 0.01 not_applicable
79 0.01 not_applicable
   climate_model dl (meter) du (meter) efficacy (dimensionless)
  0 two_layer 1200 50 1.0
  1 two_layer 1200 50 1.0
  2 two_layer 1200 50 1.0
  3 two_layer 1200 50 1.0
  4 two_layer 1200 50 1.0
.. ... ...
```
two_layer 1200 50 1.0
    eta (watt / delta_degree_Celsius / meter ** 2)
    0 0.8
    1 0.8
    2 0.8
    3 0.8
    4 0.8
    ... ...
    75 0.8
    76 0.8
    77 0.8
    78 0.8
    79 0.8

lambda0 (watt / delta_degree_Celsius / meter ** 2) mip_era model
    
    0 1.246667 CMIP6 AIM/CGE
    1 1.246667 CMIP6 AIM/CGE
    2 1.246667 CMIP6 AIM/CGE
    3 1.246667 CMIP6 AIM/CGE
    4 1.246667 CMIP6 AIM/CGE
    ... ... ...
    75 1.246667 CMIP6 REMIND-MAGPIE
    76 1.246667 CMIP6 REMIND-MAGPIE
    77 1.246667 CMIP6 REMIND-MAGPIE
    78 1.246667 CMIP6 REMIND-MAGPIE
    79 1.246667 CMIP6 REMIND-MAGPIE

region run_idx scenario unit
    0 World 0 ssp370 W/m^2
    1 World 0 ssp370 delta_degC
    2 World 0 ssp370 delta_degC
    3 World 0 ssp370 W/m^2
    4 World 1 ssp370-lowNTCF-aerchemmip W/m^2
    ... ... ...
    75 World 8 ssp534-over W/m^2
    76 World 9 ssp585 W/m^2
    77 World 9 ssp585 delta_degC
    78 World 9 ssp585 delta_degC
    79 World 9 ssp585 W/m^2

variable
    0 Effective Radiative Forcing
    1 Surface Temperature|Upper
    2 Surface Temperature|Lower
    3 Heat Uptake
    4 Effective Radiative Forcing
    ... ...
    75 Heat Uptake
    76 Effective Radiative Forcing
    77 Surface Temperature|Upper
    78 Surface Temperature|Lower
79  Heat Uptake
[80 rows x 15 columns]

[6]: # NBVAL_IGNORE_OUTPUT
pkwargs = dict(
    hue="scenario", style="a (watt / delta_degree_Celsius ** 2 / meter ** 2)",
)
fig = plt.figure(figsize=(12, 18))
ax = fig.add_subplot(211)
output.filter(variable="Surface Temperature|Upper").lineplot(**pkwargs, ax=ax)
ax = fig.add_subplot(212)
output.filter(variable="Heat Uptake").lineplot(**pkwargs, ax=ax)

[6]: <AxesSubplot:xlabel='time', ylabel='W/m^2'>
2.2 More detail

2.2.1 Impulse response equivalence

In this notebook we explore the equivalence between the two-layer model and a two-timescale impulse response approach.

Background

Following Geoffroy et al., 2013, Part 2, with notation altered to match our implementation, the two-layer model with efficacy ($\epsilon$) and state-dependent climate feedback can be written as

\[
C \frac{dT}{dt} = F - (\lambda_0 - aT)T - \epsilon\eta(T - T_D) \tag{2.1}
\]

\[
C_D \frac{dT_D}{dt} = \eta(T - T_D) \tag{2.2}
\]

If the state-dependent feedback factor, $a$, is non-zero, the two-layer model and impulse response approaches are not equivalent. However, if $a = 0$, they become the same.

Hereafter we assume $a = 0$, however this assumption should not be forgotten. In the case $a = 0$, the two-layer model can be written (adding an $\epsilon$ for the deep-ocean equation too for simplicity later).

\[
C \frac{dT}{dt} = F - \lambda_0 T - \epsilon\eta(T - T_D) \tag{2.3}
\]

\[
\epsilon C_D \frac{dT_D}{dt} = \epsilon\eta(T - T_D) \tag{2.4}
\]

In matrix notation we have

\[
\frac{dX}{dt} = AX + B \tag{2.5}
\]

where $X = \begin{pmatrix} T \\ T_D \end{pmatrix}$, $A = \begin{bmatrix} \frac{\lambda_0 + \epsilon\eta}{\epsilon C_D} & \frac{\epsilon\eta}{\epsilon C_D} \\ -\frac{\epsilon\eta}{\epsilon C_D} & -\frac{1}{\tau_2} \end{bmatrix}$ and $B = \begin{pmatrix} F \\ 0 \end{pmatrix}$.

As shown in Geoffroy et al., 2013, Part 1, $A$ can be diagonalised i.e. written in the form $A = \Phi D \Phi^{-1}$, where $D$ is a diagonal matrix. Applying the solution given in Geoffroy et al., 2013, Part 1 to our impulse response notation, we have

\[
D = \begin{bmatrix} -\frac{1}{\tau_1} & 0 \\ 0 & -\frac{1}{\tau_2} \end{bmatrix} \tag{2.6}
\]

and

\[
\Phi = \begin{bmatrix} 1 & 1 \\ \phi_1 & \phi_2 \end{bmatrix} \tag{2.7}
\]

where

\[
\tau_1 = \frac{CC_D}{2\lambda_0 \eta} (b - \sqrt{\delta}) \tag{2.8}
\]

\[
\tau_2 = \frac{CC_D}{2\lambda_0 \eta} (b + \sqrt{\delta}) \tag{2.9}
\]
\[ \phi_1 = \frac{C}{2e\eta} (b^* - \sqrt{\delta}) \]  
(2.10)

\[ \phi_2 = \frac{C}{2e\eta} (b^* + \sqrt{\delta}) \]  
(2.11)

\[ b = \frac{\lambda_0 + \epsilon\eta}{C} + \frac{\eta}{C_D} \]  
(2.12)

\[ b^* = \frac{\lambda_0 + \epsilon\eta}{C} - \frac{\eta}{C_D} \]  
(2.13)

\[ \delta = b^2 - 4 \frac{\lambda_0 \eta}{CC_D} \]  
(2.14)

Given this, we can re-write the system as

\[ \frac{dX}{dt} = \Phi \Phi^{-1} X + B \]  
(2.15)

\[ \Phi^{-1} \frac{dX}{dt} = D \Phi^{-1} X + \Phi^{-1} B \]  
(2.16)

\[ \frac{dY}{dt} = DY + \Phi^{-1} B \]  
(2.17)

(2.18)

Defining \( Y = \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} \), we have

\[ \frac{d}{dt} \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} = \begin{bmatrix} -\frac{1}{\tau_1} & 0 \\ 0 & -\frac{1}{\tau_2} \end{bmatrix} \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} + \frac{1}{\phi_2 - \phi_1} \begin{bmatrix} \phi_2 & -1 \\ -\phi_1 & 1 \end{bmatrix} \begin{bmatrix} F \\ 0 \end{bmatrix} \]  
(2.19)

or,

\[ \frac{dT_1}{dt} = \frac{\tau_1}{\tau_1} + \frac{\phi_2}{\phi_2 - \phi_1} F 
\]  
(2.20)

\[ \frac{dT_2}{dt} = \frac{\tau_2}{\tau_2} - \frac{\phi_1}{\phi_2 - \phi_1} F 
\]  
(2.21)

Re-writing, we have,

\[ \frac{dT_1}{dt} = \frac{1}{\tau_1} \left( \frac{\tau_1 \phi_2}{\phi_2 - \phi_1} F - T_1 \right) \]  
(2.22)

\[ \frac{dT_2}{dt} = \frac{1}{\tau_2} \left( -\tau_2 \phi_1 F - T_2 \right) \]  
(2.23)

We can compare this to the notation of Millar et al., 2017 and see that

\[ d_1 = \tau_1 \]  
(2.24)

\[ d_2 = \tau_2 \]  
(2.25)

\[ q_1 = \frac{\tau_1 \phi_2}{C(\phi_2 - \phi_1)} \]  
(2.26)

\[ q_2 = -\frac{\tau_2 \phi_1}{C(\phi_2 - \phi_1)} \]  
(2.27)
Hence we have redemonstrated the equivalence of the two-layer model and a two-timescale impulse response model. Given the parameters of the two-layer model, we can now trivially derive the equivalent parameters of the two-timescale model. Doing the reverse is possible, but requires some more work in order to make a usable route drop out.

The first step is to follow Geoffroy et al., 2013, Part 1, and define two extra constants

\[
a_1 = \frac{\phi_2 \tau_1}{C(\phi_2 - \phi_1)} \lambda_0 \\
a_2 = -\frac{\phi_1 \tau_2}{C(\phi_2 - \phi_1)} \lambda_0
\]  

These constants have the useful property that \(a_1 + a_2 = 1\) (proof in Appendix A).

From above, we also see that

\[
a_1 = \lambda_0 q_1 \\
a_2 = \lambda_0 q_2
\]  

Hence

\[
a_1 + a_2 = \lambda_0 q_1 + \lambda_0 q_2 = 1
\]  

\[
\lambda_0 = \frac{1}{q_1 + q_2}
\]  

Next we calculate \(C\) via

\[
\frac{q_1}{d_1} + \frac{q_2}{d_2} = \frac{\phi_2}{C(\phi_2 - \phi_1)} - \frac{\phi_1}{C(\phi_2 - \phi_1)} = \frac{1}{C}
\]  

\[
C = \frac{d_1 d_2}{q_1 d_2 + q_2 d_1}
\]  

We then use further relationships from Table 1 of Geoffroy et al., 2013, Part 1 (proof is left to the reader) to calculate the rest of the constants.

Firstly,

\[
\tau_1 a_1 + \tau_2 a_2 = \frac{C + \epsilon C_D}{\lambda_0} \\
\epsilon C_D = \lambda_0 (\tau_1 a_1 + \tau_2 a_2) - C
\]  

and then finally,

\[
\tau_1 a_1 + \tau_2 a_2 = \frac{C + \epsilon C_D}{\lambda_0} \\
\epsilon \eta = \frac{\epsilon C_D}{\tau_1 a_1 + \tau_2 a_2}
\]  

The final thing to notice here is that \(C_D\), \(\epsilon\) and \(\eta\) are not uniquely-defined. This makes sense, as shown by Geoffroy et al., 2013, Part 2, the introduction of the efficacy factor does not alter the behaviour of the system (it is still the same mathematical system) and so it is impossible for simply the two-timescale temperature response to uniquely define all three of these quantities. It can only define the products \(\epsilon C_D\) and \(\epsilon \eta\). Hence when translating from the two-timescale model to the two-layer model with efficacy, an explicit choice for the efficacy must be made. This does not alter the temperature response but it does alter the implied ocean heat uptake of the two-timescale model.

Long story short, when deriving two-layer model parameters from a two-timescale model, one must specify the efficacy.

2.2. More detail
Given that \( Y = \Phi^{-1}X \) i.e. \( X = \Phi Y \), we can also relate the impulse response boxes to the two layers.

Finally, the equivalent of the two-timescale and two-layer models allows us to also calculate the heat uptake of a two-timescale impulse response model. It is given by

\[
\text{Heat uptake} = C \frac{dT}{dt} + C_D \frac{dT_D}{dt} \\
= F - \lambda_0 T + (1 - \epsilon)\eta(T - T_D) \\
= F - \lambda_0(T_1 + T_2) + (1 - \epsilon)\eta((1 - \phi_1)T_1 + (1 - \phi_2)T_2) \\
= F - \lambda_0(T_1 + T_2) - \eta(\epsilon - 1)((1 - \phi_1)T_1 + (1 - \phi_2)T_2)
\]

**Running the code**

Here we actually run the two implementations to explore their similarity.

```python
from openscm_units.unit_registry import ur
from openscm_twolayermodel import ImpulseResponseModel, TwoLayerModel

First we define a scenario to run.

```
Next we run the two-layer model. In order for it to be convertible to a two-timescale model, we must turn state-dependence off \((a=0)\).

```
[4]: two_layer_config = {
    "du": 55 * ur("m"),
    "efficacy": 1.2 * ur("dimensionless"),
    # "efficacy": 1.0 * ur("dimensionless"),
    "a": 0 * ur("W/m^2/delta_degC^2"),
}
```

```
[5]: # NBVAL_IGNORE_OUTPUT
twolayer = TwoLayerModel(**two_layer_config)
res_twolayer = twolayer.run_scenarios(inp)
res_twolayer
```

```
HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',...
  →layout=Layout(width='20px')...)
```

```
[5]: <scmdata.ScmRun (timeseries: 4, timepoints: 751)>
Time:
  Start: 1750-01-01T00:00:00
  End: 2500-01-01T00:00:00
Meta:
  a (watt / delta_degree_Celsius ** 2 / meter ** 2) climate_model \
  0 0.0 two_layer
  1 0.0 two_layer
  2 0.0 two_layer
```

(continues on next page)
### Two Layer Model Documentation

---

#### Variables

<table>
<thead>
<tr>
<th>run_idx</th>
<th>scenario</th>
<th>unit</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>test_scenario</td>
<td>W/m^2</td>
<td>Effective Radiative Forcing</td>
</tr>
<tr>
<td>1</td>
<td>test_scenario</td>
<td>delta_degC</td>
<td>Surface Temperature</td>
</tr>
<tr>
<td>2</td>
<td>test_scenario</td>
<td>delta_degC</td>
<td>Surface Temperature</td>
</tr>
<tr>
<td>3</td>
<td>test_scenario</td>
<td>W/m^2</td>
<td>Heat Uptake</td>
</tr>
</tbody>
</table>

---

#### Example Code

```python
fig = plt.figure(figsize=(16, 9))

ax = fig.add_subplot(121)
res_twolayer.filter(variable="*Temperature*").lineplot(hue="variable", ax=ax)

ax = fig.add_subplot(122)
res_twolayer.filter(variable="Heat Uptake").lineplot(hue="variable", ax=ax)
```

---

---
Next we get the parameters with which we get the equivalent impulse response model.

```
[7]: two_timescale_paras = twolayer.get_impulse_response_parameters()
    two_timescale_paras

[7]: {'d1': 10345432.57029569 <Unit('joule / watt')>,
    'd2': 1118189193.114195 <Unit('joule / watt')>,
    'q1': 0.4465999986742509 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
    'q2': 0.3555390387589074 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
    'efficacy': 1.2 <Unit('dimensionless')>
```

```
[8]: # NBVAL_IGNORE_OUTPUT
    impulse_response = ImpulseResponseModel(**two_timescale_paras)
    res_impulse_response = impulse_response.run_scenarios(inp)
    res_impulse_response

HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
    layout=Layout(width='20px')))...
```

```
[8]: <scmdata.ScmRun (timeseries: 5, timepoints: 751)>
    Time:
      Start: 1750-01-01T00:00:00
      End: 2500-01-01T00:00:00
    Meta:
      climate_model | d1 (joule / watt) | d2 (joule / watt) |
      --- | --- | --- |
      0 two_timescale_impulse_response | 1.034543e+08 | 1.118189e+10 |
      1 two_timescale_impulse_response | 1.034543e+08 | 1.118189e+10 |
      2 two_timescale_impulse_response | 1.034543e+08 | 1.118189e+10 |
      3 two_timescale_impulse_response | 1.034543e+08 | 1.118189e+10 |
      4 two_timescale_impulse_response | 1.034543e+08 | 1.118189e+10 |
      efficacy (dimensionless) | model |
      --- | --- |
      0 | 1.2 unspecified |
```
(continues on next page)
1 1.2 unspecified
2 1.2 unspecified
3 1.2 unspecified
4 1.2 unspecified

\[q_1 \text{(delta\_degree\_Celsius} \times \text{meter} ^2 / \text{watt}) \backslash
0 \quad 0.4466
1 \quad 0.4466
2 \quad 0.4466
3 \quad 0.4466
4 \quad 0.4466
\]

\[q_2 \text{(delta\_degree\_Celsius} \times \text{meter} ^2 / \text{watt}) \text{ region run\_idx} \backslash
0 \quad 0.355539 \quad \text{World} \quad 0
1 \quad 0.355539 \quad \text{World} \quad 0
2 \quad 0.355539 \quad \text{World} \quad 0
3 \quad 0.355539 \quad \text{World} \quad 0
4 \quad 0.355539 \quad \text{World} \quad 0
\]

scenario unit variable
0 test\_scenario W/m^2 Effective Radiative Forcing
1 test\_scenario delta\_degC Surface Temperature|Box 1
2 test\_scenario delta\_degC Surface Temperature|Box 2
3 test\_scenario delta\_degC Surface Temperature
4 test\_scenario W/m^2 Heat Uptake

[9]: # NBVAL\_IGNORE\_OUTPUT
fig = plt.figure(figsize=(16, 9))

ax = fig.add_subplot(121)
res\_impulse\_response.filter(variable="*Temperature*")\text{.lineplot}(
    hue="variable", ax=ax
)

ax = fig.add_subplot(122)
res\_impulse\_response.filter(variable="Heat Uptake")\text{.lineplot}(
    hue="variable", ax=ax
)

[9]: <AxesSubplot:xlabel='time', ylabel='W/m^2'>
We can compare the two responses as well.

```python
combined = run_append([res_impulse_response, res_twolayer])
```

```sql
<scmdata.ScmRun (timeseries: 9, timepoints: 751)>
Time:
Start: 1750-01-01T00:00:00
End: 2500-01-01T00:00:00
Meta:

<table>
<thead>
<tr>
<th>a</th>
<th>(watt / delta_degree_Celsius ** 2 / meter ** 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>climate_model</th>
<th>d1 (joule / watt)</th>
<th>d2 (joule / watt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>two_timescale_impulse_response</td>
<td>1.034543e+08</td>
<td>1.118189e+10</td>
</tr>
<tr>
<td>two_timescale_impulse_response</td>
<td>1.034543e+08</td>
<td>1.118189e+10</td>
</tr>
<tr>
<td>two_timescale_impulse_response</td>
<td>1.034543e+08</td>
<td>1.118189e+10</td>
</tr>
<tr>
<td>two_timescale_impulse_response</td>
<td>1.034543e+08</td>
<td>1.118189e+10</td>
</tr>
<tr>
<td>two_timescale_impulse_response</td>
<td>1.034543e+08</td>
<td>1.118189e+10</td>
</tr>
<tr>
<td>two_layer</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>two_layer</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>two_layer</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>two_layer</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th>dl (meter)</th>
<th>du (meter)</th>
<th>efficacy (dimensionless)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NaN</td>
<td>1.2</td>
</tr>
<tr>
<td>1</td>
<td>NaN</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>NaN</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>NaN</td>
<td>1.2</td>
</tr>
<tr>
<td>5</td>
<td>1200.0</td>
<td>55.0</td>
</tr>
<tr>
<td>6</td>
<td>1200.0</td>
<td>55.0</td>
</tr>
<tr>
<td>7</td>
<td>1200.0</td>
<td>55.0</td>
</tr>
<tr>
<td>8</td>
<td>1200.0</td>
<td>55.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eta (watt / delta_degree_Celsius / meter ** 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lambda0 (watt / delta_degree_Celsius / meter ** 2) model</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q1 (delta_degree_Celsius * meter ** 2 / watt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q2 (delta_degree_Celsius * meter ** 2 / watt) region run_idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>scenario</th>
<th>unit</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 test_scenario</td>
<td>W/m^2 Effective Radiative Forcing</td>
<td></td>
</tr>
</tbody>
</table>
To within numerical errors they are equal.

```
[11]: # NBVAL_IGNORE_OUTPUT
fig = plt.figure(figsize=(16, 9))

ax = fig.add_subplot(121)
combined.filter(variable="*Temperature*").lineplot(
    hue="variable", style="climate_model", alpha=0.7, linewidth=2, ax=ax
)
ax.legend(loc="upper left")

ax = fig.add_subplot(122)
combined.filter(variable="Heat Uptake").lineplot(
    hue="climate_model", style="climate_model", alpha=0.7, linewidth=2, ax=ax
)
```

![Graphs showing temperature and heat uptake over time with legends for different scenarios.](attachment:image.png)

2.2. More detail
Appendix A

We begin with the definitions of the \( a \) constants,

\[
\begin{align*}
a_1 &= \frac{\phi_2 \tau_1}{C(\phi_2 - \phi_1)} \lambda_0 \\ a_2 &= -\frac{\phi_1 \tau_2}{C(\phi_2 - \phi_1)} \lambda_0
\end{align*}
\]

We then have

\[
\begin{align*}
a_1 + a_2 &= \frac{\phi_2 \tau_1}{C(\phi_2 - \phi_1)} \lambda_0 - \frac{\phi_1 \tau_2}{C(\phi_2 - \phi_1)} \lambda_0 \\ &= \frac{\lambda_0}{C(\phi_2 - \phi_1)} (\phi_2 \tau_1 - \phi_1 \tau_2)
\end{align*}
\]

Recalling the definition of the \( \phi \) parameters,

\[
\begin{align*}
\phi_1 &= \frac{C}{2\epsilon \eta} (b^* - \sqrt{\delta}) \\ \phi_2 &= \frac{C}{2\epsilon \eta} (b^* + \sqrt{\delta})
\end{align*}
\]

We have,

\[
\phi_2 - \phi_1 = \frac{C \sqrt{\delta}}{\epsilon \eta}
\]

Recalling the definition of the \( \tau \) parameters,

\[
\begin{align*}
\tau_1 &= \frac{CC_D}{2\lambda_0 \eta} (b - \sqrt{\delta}) \\ \tau_2 &= \frac{CC_D}{2\lambda_0 \eta} (b + \sqrt{\delta})
\end{align*}
\]

We have,

\[
\phi_2 \tau_1 - \phi_1 \tau_2 = \frac{C}{2\epsilon \eta} (b^* + \sqrt{\delta}) \times \frac{CC_D}{2\lambda_0 \eta} (b - \sqrt{\delta}) - \frac{C}{2\epsilon \eta} (b^* - \sqrt{\delta}) \times \frac{CC_D}{2\lambda_0 \eta} (b + \sqrt{\delta})
\]

\[
= \frac{C^2 C_d}{4\epsilon \eta^2 \lambda_0} \left[ (b^* + \sqrt{\delta})(b - \sqrt{\delta}) - (b^* - \sqrt{\delta})(b + \sqrt{\delta}) \right]
\]

\[
= \frac{C^2 C_d}{4\epsilon \eta^2 \lambda_0} \left[ b^* b + b \sqrt{\delta} - b^* \sqrt{\delta} - \delta - bb^* + b \sqrt{\delta} - b^* \sqrt{\delta} + \delta \right]
\]

\[
= \frac{C^2 C_d}{2\epsilon \eta^2 \lambda_0} \left[ \sqrt{\delta} - b^* \sqrt{\delta} \right]
\]

\[
= \frac{C^2 C_d \sqrt{\delta}}{2\epsilon \eta^2 \lambda_0} \left[ b - b^* \right]
\]

Recalling the definition of the \( b \) parameters,

\[
b = \frac{\lambda_0 + \epsilon \eta}{C} + \frac{\eta}{C_D}
\]
\[ b^* = \frac{\lambda_0 + \eta_n}{C} - \frac{\eta}{C_D} \]  

(2.59)

We then have

\[ \phi_2 \tau_1 - \phi_1 \tau_2 = \frac{C^2 C_d \sqrt{\delta}}{2 \eta \lambda_0} \left[ \frac{2 \eta}{C_D} \right] \]

(2.60)

\[ = \frac{C^2 \sqrt{\delta}}{c \eta \lambda_0} \]  

(2.61)

Putting it all back together,

\[ a_1 + a_2 = \frac{\lambda_0}{C (\phi_2 - \phi_1)} (\phi_2 \tau_1 - \phi_1 \tau_2) \]

(2.62)

\[ = \frac{\lambda_0}{C} \frac{\eta}{C^2 \sqrt{\delta}} \]

(2.63)

\[ = 1 \]  

(2.64)

### 2.2.2 One layer model

Here we show how to run our two-layer model as a single-layer model. Put simply, we make the deep ocean component of the two-layer model infinitely deep and we have a single layer model. The concept is described by Equation 4 of Geoffroy et al. 2013, Part 1.

```python
# NBVAL_IGNORE_OUTPUT
import os.path
import numpy as np
import pandas as pd
import openscm_units.unit_registry as ur
import tqdm.autonotebook as tqdman
from scmdata import ScmRun, run_append
from openscm_twolayermodel import TwoLayerModel
import matplotlib.pyplot as plt

DATA_PATH = os.path.join('..', '..', '..', 'tests', 'test-data', 'rcmip-radiative-forcing-annual-means-v4-0-0.csv')

For this we use RCPMIP effective radiative forcing data.
```

```python
DATA_PATH = os.path.join('..', '..', '..', 'tests', 'test-data', 'rcmip-radiative-forcing-annual-means-v4-0-0.csv')
```

2.2. More detail
```python
scenarios = ScmRun(DATA_PATH, lowercase_cols=True).filter(scenario="historical", keep=False)
scenarios
```

```
<scmdata.ScmRun (timeseries: 480, timepoints: 751)>
Time:
    Start: 1750-01-01T00:00:00
    End: 2500-01-01T00:00:00
Meta:
    activity_id mip_era model region scenario unit
0 not_applicable CMIP5 AIM World rcp60 W/m^2
1 not_applicable CMIP5 AIM World rcp60 W/m^2
2 not_applicable CMIP5 AIM World rcp60 W/m^2
3 not_applicable CMIP5 AIM World rcp60 W/m^2
4 not_applicable CMIP5 AIM World rcp60 W/m^2
...
494 not_applicable CMIP5 unspecified World historical-cmip5 W/m^2
495 not_applicable CMIP5 unspecified World historical-cmip5 W/m^2
496 not_applicable CMIP5 unspecified World historical-cmip5 W/m^2
497 not_applicable CMIP5 unspecified World historical-cmip5 W/m^2
498 not_applicable CMIP5 unspecified World historical-cmip5 W/m^2

variable
0 Radiative Forcing
1 Radiative Forcing|Anthropogenic
2 Radiative Forcing|Anthropogenic|Aerosols
3 Radiative Forcing|Anthropogenic|Aerosols|Aeros...
4 Radiative Forcing|Anthropogenic|Aerosols|Aeros...
...
494 Radiative Forcing|Anthropogenic|Stratospheric ...
495 Radiative Forcing|Anthropogenic|Tropospheric O...
496 Radiative Forcing|Natural
497 Radiative Forcing|Natural|Solar
498 Radiative Forcing|Natural|Volcanic
```

```
fig = plt.figure(figsize=(12, 18))
ax = fig.add_subplot(111)
scenarios.filter(variable="Effective Radiative Forcing").lineplot()
```

```
<AxesSubplot:xlabel='time', ylabel='W/m^2'>
```
2.2. More detail
dl_values = np.array([100, 2000, 10000, 10**15]) * ur("m")

runner = TwoLayerModel()
output = []
equivalent_parameters = []
for dl in tqdm(dl_values, desc="Parameter settings"):
    runner.dl = dl
    output.append(runner.run_scenarios(scenarios))
equivalent_parameters.append((("two-layer deep ocean depth": runner.dl), runner.
    \get_impulse_response_parameters()))
output = run_append(output)
output

HBox(children=(HTML(value='Parameter settings'), FloatProgress(value=0.0, max=4.0),
    \HTML(value='')))

HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
    \layout=Layout(width='20px'))...)

HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
    \layout=Layout(width='20px'))...)

HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
    \layout=Layout(width='20px'))...)

HBox(children=(HTML(value='scenarios'), FloatProgress(value=1.0, bar_style='info',
    \layout=Layout(width='20px'))...)

<scmdata.ScmRun (timeseries: 160, timepoints: 751)>
Time:
  Start: 1750-01-01T00:00:00
  End: 2500-01-01T00:00:00
Meta:
   a (watt / delta_degree_Celsius ** 2 / meter ** 2) activity_id \
  0  0.0 not_applicable
  1  0.0 not_applicable
  2  0.0 not_applicable
  3  0.0 not_applicable
  4  0.0 not_applicable
   ..  ... ... ... ...
 155  0.0 not_applicable
 156  0.0 not_applicable
 157  0.0 not_applicable
 158  0.0 not_applicable
 159  0.0 not_applicable

   climate_model dl (meter) du (meter) efficacy (dimensionless) \
  0 two_layer 100 50 1.0
  1 two_layer 100 50 1.0
  2 two_layer 100 50 1.0
  3 two_layer 100 50 1.0
  4 two_layer 100 50 1.0
   .. ... ... ... ...

(continues on next page)
| two_layer | 1000000000000000 | 50 | 1.0 |
| two_layer | 1000000000000000 | 50 | 1.0 |
| two_layer | 1000000000000000 | 50 | 1.0 |
| two_layer | 1000000000000000 | 50 | 1.0 |
| two_layer | 1000000000000000 | 50 | 1.0 |

\[
\text{eta} \ (\text{watt} / \text{delta}\_\text{degree}\_\text{Celsius} / \text{meter}^{\ast} 2) \ \backslash
\]
0 
1 
2 
3 
4 
.. 
155 
156 
157 
158 
159

\[
\text{lambda0} \ (\text{watt} / \text{delta}\_\text{degree}\_\text{Celsius} / \text{meter}^{\ast} 2) \ \text{mip}\_\text{era} \ \backslash
\]
0 
1 
2 
3 
4 
.. 
155 
156 
157 
158 
159

<table>
<thead>
<tr>
<th>model</th>
<th>region</th>
<th>run_idx</th>
<th>scenario</th>
<th>unit</th>
</tr>
</thead>
</table>
0 | AIM/CGE | World | 0 | ssp370 | W/m^2 |
1 | AIM/CGE | World | 0 | ssp370 | delta_degC |
2 | AIM/CGE | World | 0 | ssp370 | delta_degC |
3 | AIM/CGE | World | 0 | ssp370 | W/m^2 |
4 | AIM/CGE | World | 1 | ssp370-lowNTCF-aerchemmmip | W/m^2 |
.. | ... | ... | ... | ... |
155 | REMIND-MAGPIE | World | 8 | ssp534-over | W/m^2 |
156 | REMIND-MAGPIE | World | 9 | ssp585 | W/m^2 |
157 | REMIND-MAGPIE | World | 9 | ssp585 | delta_degC |
158 | REMIND-MAGPIE | World | 9 | ssp585 | delta_degC |
159 | REMIND-MAGPIE | World | 9 | ssp585 | W/m^2 |

variable
0 | Effective Radiative Forcing |
1 | Surface Temperature|Upper |
2 | Surface Temperature|Lower |
3 | Heat Uptake |
4 | Effective Radiative Forcing |
.. | ... |
155 | Heat Uptake |
156 | Effective Radiative Forcing |
157 | Surface Temperature|Upper |
158 | Surface Temperature|Lower |
159 | Heat Uptake |
As we can see in the plots below, as the deep ocean becomes bigger, it can uptake more heat and hence warming is reduced. Related to this, we also see that the deeper the deep ocean is, the longer the equilibration time (but the harder it is to see on the time axis we have). This is because the slow timescale becomes too long for us to see it i.e. we effectively have only one timescale in the model (as discussed in Geoffroy et al. 2013, Part I). This is clear if we look at the slow-timescale in the equivalent two-timescale response parameters. As the deep ocean becomes bigger and bigger, the slow-timescale grows from hundreds to thousands to billions of years.

```python
[7]: for v in equivalent_parameters:
    v[1]['d1'] = v[1]['d1'].to('yr')
    v[1]['d2'] = v[1]['d2'].to('yr')

[7]: [({'two-layer deep ocean depth': 100 <Unit('meter')>},
    {'d1': 2.980911326367698 <Unit('a')>,
     'd2': 29.52113470568794 <Unit('a')>,
     'q1': 0.4104377535043735 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'q2': 0.3917012839287813 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'efficacy': 1.0 <Unit('dimensionless')>}),
   ({'two-layer deep ocean depth': 2000 <Unit('meter')>},
    {'d1': 3.224285795910259 <Unit('a')>,
     'd2': 54.58565630896654 <Unit('a')>,
     'q1': 0.4848549694209551 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'q2': 0.3172840680120327 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'efficacy': 1.0 <Unit('dimensionless')>}),
   ({'two-layer deep ocean depth': 10000 <Unit('meter')>},
    {'d1': 3.2342013046041056 <Unit('a')>,
     'd2': 272.91530058768 <Unit('a')>,
     'q1': 0.4878523568580023 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'q2': 0.3142866805751838 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'efficacy': 1.0 <Unit('dimensionless')>}),
   ({'two-layer deep ocean depth': 1000000000000000 <Unit('meter')>},
    {'d1': 3.23895349323281 <Unit('a')>,
     'd2': 27183935162560.66 <Unit('a')>,
     'q1': 0.4889441930744013 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'q2': 0.3155597245418723 <Unit('delta_degree_Celsius * meter ** 2 / watt')>,
     'efficacy': 1.0 <Unit('dimensionless')>})]
```

```python
[8]: # NBVAL_IGNORE_OUTPUT
    pkwargs = dict(
        hue="scenario", style="{} (meter)"
    )
    fig = plt.figure(figsize=(12, 18))
    ax = fig.add_subplot(211)
    output.filter(scenario="ssp585", variable="Surface Temperature|Upper").
    lineplot(**pkwargs, ax=ax)
    ax = fig.add_subplot(212)
    output.filter(scenario="ssp585", variable="Heat Uptake").lineplot(**pkwargs, ax=ax)
[8]: <AxesSubplot:xlabel='time', ylabel='W/m^2'>
```
2.2. More detail
If you’re interested in contributing to OpenSCM Two Layer Model, we’d love to have you on board! This section of the docs will (once we’ve written it) detail how to get setup to contribute and how best to communicate.

- Contributing
- Getting setup
  - Getting help
    * Development tools
    * Other tools
- Formatting
- Building the docs
  - Gotchas
  - Docstring style
- Releasing
  - First step
  - Push to repository
- Why is there a Makefile in a pure Python repository?

### 3.1 Contributing

All contributions are welcome, some possible suggestions include:

- tutorials (or support questions which, once solved, result in a new tutorial :D)
- blog posts
- improving the documentation
- bug reports
- feature requests
- pull requests

Please report issues or discuss feature requests in the OpenSCM Two Layer Model issue tracker. If your issue is a feature request or a bug, please use the templates available, otherwise, simply open a normal issue :)

47
As a contributor, please follow a couple of conventions:

- Create issues in the OpenSCM Two Layer Model issue tracker for changes and enhancements, this ensures that everyone in the community has a chance to comment.
- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds: see the Python Community Code of Conduct.
- Only push to your own branches, this allows people to force push to their own branches as they need without fear or causing others headaches.
- Start all pull requests as draft pull requests and only mark them as ready for review once they’ve been rebased onto master, this makes it much simpler for reviewers.
- Try and make lots of small pull requests, this makes it easier for reviewers and faster for everyone as review time grows exponentially with the number of lines in a pull request.

### 3.2 Getting setup

To get setup as a developer, we recommend the following steps (if any of these tools are unfamiliar, please see the resources we recommend in Development tools):

1. Install conda and make
2. Run `make virtual-environment`, if that fails you can try doing it manually.
   1. Change your current directory to OpenSCM Two Layer Model’s root directory (i.e. the one which contains `README.rst`), `cd openscm-twolayermodel`
   2. Create a virtual environment to use with OpenSCM Two Layer Model `python3 -m venv venv`
   3. Activate your virtual environment `source ./venv/bin/activate`
   4. Upgrade pip `pip install --upgrade pip`
   5. Install the development dependencies (very important, make sure your virtual environment is active before doing this) `pip install -e .[dev]`
3. Make sure the tests pass by running `make check`, if that fails the commands can be read out of the `Makefile`.

### 3.2.1 Getting help

Whilst developing, unexpected things can go wrong (that’s why it’s called ‘developing’, if we knew what we were doing, it would already be ‘developed’). Normally, the fastest way to solve an issue is to contact us via the issue tracker. The other option is to debug yourself. For this purpose, we provide a list of the tools we use during our development as starting points for your search to find what has gone wrong.

**Development tools**

This list of development tools is what we rely on to develop OpenSCM Two Layer Model reliably and reproducibly. It gives you a few starting points in case things do go inexplicably wrong and you want to work out why. We include links with each of these tools to starting points that we think are useful, in case you want to learn more.

- Git
- Make
- Conda virtual environments
• Pip and pip virtual environments

• Tests
  – we use a blend of pytest and the inbuilt Python testing capabilities for our tests so checkout what we’ve
    already done in tests to get a feel for how it works

• Continuous integration (CI) (also brief intro blog post and a longer read)
  – we use GitHub CI for our CI but there are a number of good providers

• Jupyter Notebooks
  – Jupyter is automatically included in your virtual environment if you follow our Getting setup instructions

• Sphinx

Other tools

We also use some other tools which aren’t necessarily the most familiar. Here we provide a list of these along with
useful resources.

• Regular expressions
  – we use regex101.com to help us write and check our regular expressions, make sure the language is set to
    Python to make your life easy!

3.3 Formatting

To help us focus on what the code does, not how it looks, we use a couple of automatic formatting tools. These
automatically format the code for us and tell use where the errors are. To use them, after setting yourself up (see
Getting setup), simply run make format. Note that make format can only be run if you have committed all
your work i.e. your working directory is ‘clean’. This restriction is made to ensure that you don’t format code without
being able to undo it, just in case something goes wrong.

3.4 Building the docs

After setting yourself up (see Getting setup), building the docs is as simple as running make docs (note, run make
-B docs to force the docs to rebuild and ignore make when it says ‘... index.html is up to date’). This will build the
docs for you. You can preview them by opening docs/build/html/index.html in a browser.

For documentation we use Sphinx. To get ourselves started with Sphinx, we started with this example then used
Sphinx’s getting started guide.

3.4.1 Gotchas

To get Sphinx to generate pdfs (rarely worth the hassle), you require LaTeXmk. On a Mac this can be installed
with sudo tlmgr install latexmk. You will most likely also need to install some other packages (if you don’t have the full
distribution). You can check which package contains any missing files with tlmgr
search --global --file [filename]. You can then install the packages with sudo tlmgr install
[package].
3.4.2 Docstring style

For our docstrings we use numpy style docstrings. For more information on these, here is the full guide and the quick reference we also use.

3.5 Releasing

3.5.1 First step

1. Test installation with dependencies `make test-install`
2. Update `CHANGELOG.rst`
   - add a header for the new version between `master` and the latest bullet point
   - this should leave the section underneath the master header empty
3. `git add .`
4. `git commit -m "Prepare for release of vX.Y.Z"`
5. `git tag vX.Y.Z`
6. Test version updated as intended with `make test-install`

3.5.2 Push to repository

To do the release, push the tags and the repository state.

1. `git push`
2. `git push --tags`

Assuming all the checks pass, this automatically triggers a release on PyPI via the `.github/workflows/ci-cd-workflow.yml` action.

3.6 Why is there a `Makefile` in a pure Python repository?

Whilst it may not be standard practice, a `Makefile` is a simple way to automate general setup (environment setup in particular). Hence we have one here which basically acts as a notes file for how to do all those little jobs which we often forget e.g. setting up environments, running tests (and making sure we’re in the right environment), building docs, setting up auxillary bits and pieces.
Module containing the base for model implementations

```python
class openscm_twolayermodel.base.Model
    Bases: abc.ABC

    Base class for model implementations

    reset()
        Reset everything so that a new run can be performed.
        Called as late as possible before run().

    run()
        Run the model.

    abstract set_drivers(*args, **kwargs)
        Set the model’s drivers

    step()
        Do a single time step.
```

```python
class openscm_twolayermodel.base.TwoLayerVariant
    Bases: openscm_twolayermodel.base.Model

    Base for variations of implementations of the two-layer model

    property delta_t
        pint.Quantity Time step for forward-differencing approximation

    property erf
        pint.Quantity Effective radiative forcing

    reset()
        Reset everything so that a new run can be performed.
        Called as late as possible before run().

    run()
        Run the model.

    run_scenarios(scenarios, driver_var='Effective Radiative Forcing', progress=True)
        Run scenarios.

        The model timestep is automatically adjusted based on the timestep used in scenarios. The timestep used in scenarios must be constant because this implementation has a constant timestep. Pull requests to upgrade the implementation to support variable timesteps are welcome https://github.com/openscm/openscm-twolayermodel/pulls.

        Parameters
```

```python```
- **scenarios** *(ScmDataFrame or ScmRun or pyam.IamDataFrame or pd.DataFrame or np.ndarray or str)* – Scenarios to run. The input will be converted to an ScmRun before the run takes place.

- **driver_var** *(str)* – The variable in scenarios to use as the driver of the model

- **progress** *(bool)* – Whether to display a progress bar

**Returns** Results of the run (including drivers)

**Return type** ScmRun

**Raises** `ValueError` – No data is available for `driver_var` in the "World" region in scenarios.

```python
def set_drivers(erf):
    """Set drivers for a model run""

    Parameters:
    erf (pint.Quantity) – Effective radiative forcing (W/m^2) to use to drive the model

    Raises `AssertionError` – erf is not one-dimensional
```

```python
def step():
    """Do a single time step.""
```
Module containing the impulse response model

The 2-timescale impulse response model is mathematically equivalent to the two-layer model without state dependence.

```python
class openscm_twolayermodel.impulse_response_model.ImpulseResponseModel(q1=<Quantity(0.3, 'delta_degree_Celsius * meter ** 2 / watt'>,
q2=<Quantity(0.4, 'delta_degree_Celsius * meter ** 2 / watt'>,
d1=<Quantity(9.0, 'a'>,
d2=<Quantity(400.0, 'a'>,
efficiency=<Quantity(1.0, 'dimensionless'>,
delta_t=<Quantity(0.0833333333, 'a'>)
```

**Bases:** `openscm_twolayermodel.base.TwoLayerVariant`

**TODO:** top line and paper references

This implementation uses a forward-differencing approach. This means that temperature and ocean heat uptake values are start of timestep values. For example, temperature[i] is only affected by drivers from the i-1 timestep. In practice, this means that the first temperature and ocean heat uptake values will always be zero and the last value in the input drivers has no effect on model output.
property d1
   pint.Quantity Response timescale of first box

property d2
   pint.Quantity Response timescale of second box

property delta_t
   pint.Quantity Time step for forward-differencing approximation

property efficacy
   pint.Quantity Efficacy factor

property erf
   pint.Quantity Effective radiative forcing

get_two_layer_parameters()
   Get equivalent two-layer model parameters
   For details on how the equivalence is calculated, please see the notebook
   impulse-response-equivalence.ipynb in the OpenSCM Two Layer model repository.
   Returns dict of str – Input arguments to initialise an openscm_twolayermodel.TwoLayerModel
   with the same temperature response as self
   Return type pint.Quantity

property q1
   pint.Quantity Sensitivity of first box response to radiative forcing

property q2
   pint.Quantity Sensitivity of second box response to radiative forcing

reset()
   Reset everything so that a new run can be performed.
   Called as late as possible before run().

run()
   Run the model.

run_scenarios(scenarios, driver_var='Effective Radiative Forcing', progress=True)
   Run scenarios.
   The model timestep is automatically adjusted based on the timestep used in scenarios. The timestep
   used in scenarios must be constant because this implementation has a constant timestep. Pull requests
   to upgrade the implementation to support variable timesteps are welcome https://github.com/openscm/
   openscm-twolayermodel/pulls.
   Parameters
      • scenarios (ScmDataFrame or ScmRun or pyam.IamDataFrame or pd.
        DataFrame or np.ndarray or str) – Scenarios to run. The input will be converted
to an ScmRun before the run takes place.
      • driver_var(str) – The variable in scenarios to use as the driver of the model
      • progress (bool) – Whether to display a progress bar
   Returns Results of the run (including drivers)
   Return type ScmRun
   Raises ValueError – No data is available for driver_var in the "World" region in
   scenarios.
**set_drivers**(*erf*)

Set drivers for a model run

**Parameters**

- **erf** *(pint.Quantity)* – Effective radiative forcing (W/m^2) to use to drive the model

**Raises** *AssertionError* – *erf* is not one-dimensional

**step**()

Do a single time step.
Module containing the two-layer model

```python
class openscm_twolayermodel.two_layer_model.TwoLayerModel (du=<Quantity(50, 'meter')>,
dl=<Quantity(1200, 'meter')>,
lambda0=<Quantity(1.24666667, watt / delta_degree_Celsius / meter ** 2)>,
a=<Quantity(0.0, watt / delta_degree_Celsius ** 2 / meter ** 2)>,
efficacy=<Quantity(1.0, 'dimensionless')>,
eta=<Quantity(0.8, watt / delta_degree_Celsius / meter ** 2)>,
delta_t=<Quantity(31557600.0, 'second')>)
```

Bases: `openscm_twolayermodel.base.TwoLayerVariant`

TODO: top line and paper references

This implementation uses a forward-differencing approach. This means that temperature and ocean heat uptake values are start of timestep values. For example, temperature[i] is only affected by drivers from the i-1 timestep. In practice, this means that the first temperature and ocean heat uptake values will always be zero and the last value in the input drivers has no effect on model output.

**property a**
```
pint.Quantity Dependence of climate feedback factor on temperature
```

**property delta_t**
```
pint.Quantity Time step for forward-differencing approximation
```

**property dl**
```
pint.Quantity Depth of lower layer
```

**property du**
```
pint.Quantity Depth of upper layer
```

**property efficacy**
```
pint.Quantity Efficacy factor
```
property erf
    pint.Quantity Effective radiative forcing

property eta
    pint.Quantity Heat transport efficiency

def get_impulse_response_parameters()
    Get equivalent two-timescale impulse response model parameters

    For details on how the equivalence is calculated, please see the notebook
    impulse-response-equivalence.ipynb in the OpenSCM Two Layer model repository.

    Returns dict of str – Input arguments to initialise an openscm_twolayermodel.
    ImpulseResponseModel with the same temperature response as self

    Return type pint.Quantity

    Raises ValueError – self.a is non-zero, the two-timescale model does not support state-
dependence.

property heat_capacity_lower
    pint.Quantity Heat capacity of lower layer

property heat_capacity_upper
    pint.Quantity Heat capacity of upper layer

property lambda0
    pint.Quantity Initial climate feedback factor

def reset()
    Reset everything so that a new run can be performed.
    Called as late as possible before run().

def run()
    Run the model.

def run_scenarios(scenarios, driver_var='Effective Radiative Forcing', progress=True)
    Run scenarios.

    The model timestep is automatically adjusted based on the timestep used in scenarios. The timestep
    used in scenarios must be constant because this implementation has a constant timestep. Pull requests
to upgrade the implementation to support variable timesteps are welcome https://github.com/openscm/
openscm-twolayermodel/pulls.

    Parameters

    * scenarios (ScmDataFrame or ScmRun or pyam.IamDataFrame or pd.
      DataFrame or np.ndarray or str) – Scenarios to run. The input will be converted
to an ScmRun before the run takes place.

    * driver_var (str) – The variable in scenarios to use as the driver of the model

    * progress (bool) – Whether to display a progress bar

    Returns Results of the run (including drivers)

    Return type ScmRun

    Raises ValueError – No data is available for driver_var in the "World" region in
scenarios.

def set_drivers(erf)
    Set drivers for a model run
Parameters `erf` (pint.Quantity) – Effective radiative forcing (W/m^2) to use to drive the model

Raises `AssertionError` – erf is not one-dimensional

`step()`  
Do a single time step.
Physical constants used in calculations

`s.`

`openscm_twolayermodel.constants.DENSITY_WATER = <Quantity(1000.0, 'kilogram / meter ** 3')>`

density of water

`Type pint.Quantity`

`openscm_twolayermodel.constants.HEAT_CAPACITY_WATER = <Quantity(4181.0, 'joule / delta_degree_Celsius / kilogram')>`

heat capacity of water

`Type pint.Quantity`
Errors API

Exceptions raised within openscm_twolayermodel

```python
exception openscm_twolayermodel.errors.ModelStateError
    Bases: ValueError

    Exception raised if a model’s state is incompatible with the action

    args

    with_traceback()  
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```

```python
exception openscm_twolayermodel.errors.UnitError
    Bases: ValueError

    Exception raised if something has the wrong units

    args

    with_traceback()  
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```
Utility functions

`opencm_twolayermodel.utils.convert_lambda_to ecs(lambda_val, f2x=`<Quantity`(3.74, 'watt / meter ** 2')>)`

Convert a lambda value to equilibrium climate sensitivity (ECS)

**Parameters**

- `lambda_val` (pint.Quantity) – Value of lambda to convert to ECS
- `f2x` (pint.Quantity) – Value of the forcing due to a doubling of atmospheric CO₂ to assume during the conversion

**Returns** ECS value

**Return type** pint.Quantity

**Raises** `TypeError` – `lambda_val` or `f2x` is not a `pint.Quantity`. 
All notable changes to this project will be documented in this file.
The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.
The changes listed in this file are categorised as follows:

- Added: new features
- Changed: changes in existing functionality
- Deprecated: soon-to-be removed features
- Removed: now removed features
- Fixed: any bug fixes
- Security: in case of vulnerabilities.

10.1 master

10.1.1 Changed

- (#23) Moved notebooks into full documentation following JOSS review (closes #17)
- (#21) Quoted pip install instructions to ensure cross-shell compatibility following JOSS review (closes #16)
- (#20) Option to remove tqdm progress bar by passing progress=False

10.2 v0.2.0 - 2020-10-09

10.2.1 Added

- (#7) JOSS paper draft
10.2.2 Changed

• (#7) Require `scmdata>=0.7`

10.3 v0.1.2 - 2020-03-07

10.3.1 Changed

• (#12) Upgrade to `scmdata>=0.6.2` so that package can be installed

10.4 v0.1.1 - 2020-06-29

10.4.1 Added

• (#8) Add notebook showing how to run a single-layer model

10.4.2 Changed

• (#11) Re-wrote the getting started notebook
• (#10) Re-wrote CHANGELOG
• (#9) Update to `scmdata 0.5.Y`

10.5 v0.1.0 - 2020-05-15

10.5.1 Added

• (#3) Add first implementation of the models
• (#1) Setup repository
INDEX

- genindex
- modindex
- search
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>openscm_twolayermodel.base</td>
<td>51</td>
</tr>
<tr>
<td>openscm_twolayermodel.constants</td>
<td>61</td>
</tr>
<tr>
<td>openscm_twolayermodel.errors</td>
<td>63</td>
</tr>
<tr>
<td>openscm_twolayermodel.impulse_response_model</td>
<td>53</td>
</tr>
<tr>
<td>openscm_twolayermodel.two_layer_model</td>
<td>57</td>
</tr>
<tr>
<td>openscm_twolayermodel.utils</td>
<td>65</td>
</tr>
</tbody>
</table>
INDEX

A
a() (openscm_twolayermodel.two_layer_model.TwoLayerModel
property), 57
args (openscm_twolayermodel.errors.ModelStateError
attribute), 63
args (openscm_twolayermodel.errors.UnitError attribute), 63

C
convert_lambda_to_ecs() (in module open-
scm_twolayermodel.utils), 65

D
d1() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 53
d2() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 54
delta_t() (openscm_twolayermodel.base.TwoLayerVariant
property), 51
delta_t() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 54
delta_t() (openscm_twolayermodel.two_layer_model.TwoLayerModel
property), 57
DENSITY_WATER (in module open-
scm_twolayermodel.constants), 61
d1() (openscm_twolayermodel.two_layer_model.TwoLayerModel
property), 57
du() (openscm_twolayermodel.two_layer_model.TwoLayerModel
property), 57

efficacy() (openscm_twolayermodel.impulse_response_model.
ImpulseResponseModel
property), 54
efficacy() (openscm_twolayermodel.impulse_response_model.
ImpulseResponseModel
property), 54
erf() (openscm_twolayermodel.base.TwoLayerVariant
property), 51
erf() (openscm_twolayermodel.base.TwoLayerVariant
property), 51
erf() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 54
erf() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 54
erf() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel
property), 54
eta() (openscm_twolayermodel.two_layer_model.TwoLayerModel
property), 58

G
get_impulse_response_parameters() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
method), 58
get_two_layer_parameters() (open-
scm_twolayermodel.impulse_response_model.ImpulseResponseModel
method), 54

H
heat_capacity_lower() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
heat_capacity_upper() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
HEAT_CAPACITY_WATER (in module open-
scm_twolayermodel.constants), 61

I
ImpulseResponseModel (class in open-
scm_twolayermodel.impulse_response_model), 53

L
ML

M
efficacy() (openscm_twolayermodel.impulse_response_model.
Model
property), 51
erf() (openscm_twolayermodel.impulse_response_model.
Model
property), 51
ModelStateError, 63

H
heat_capacity_lower() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
heat_capacity_upper() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
HEAT_CAPACITY_WATER (in module open-
scm_twolayermodel.constants), 61

I
ImpulseResponseModel (class in open-
scm_twolayermodel.impulse_response_model), 53

L
ML

M
efficacy() (openscm_twolayermodel.impulse_response_model.
Model
property), 51
erf() (openscm_twolayermodel.impulse_response_model.
Model
property), 51
ModelStateError, 63

H
heat_capacity_lower() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
heat_capacity_upper() (open-
scm_twolayermodel.two_layer_model.TwoLayerModel
property), 58
HEAT_CAPACITY_WATER (in module open-
scm_twolayermodel.constants), 61

I
ImpulseResponseModel (class in open-
scm_twolayermodel.impulse_response_model), 53

L
ML

M
OpenSCM Two Layer Model Documentation, Release 0.2.0+25.ge87e2fb.dirty

O
openscm_twolayermodel.base module, 51
openscm_twolayermodel.constants module, 61
openscm_twolayermodel.errors module, 63
openscm_twolayermodel.impulse_response_model module, 53
openscm_twolayermodel.two_layer_model module, 57
openscm_twolayermodel.utils module, 65

Q
q1() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel property), 54
q2() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel property), 54

R
reset() (openscm_twolayermodel.base.Model method), 51
reset() (openscm_twolayermodel.base.TwoLayerVariant method), 51
reset() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel method), 54
reset() (openscm_twolayermodel.two_layer_model.TwoLayerModel method), 58
run() (openscm_twolayermodel.base.Model method), 51
run() (openscm_twolayermodel.base.TwoLayerVariant method), 51
run() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel method), 54
run() (openscm_twolayermodel.two_layer_model.TwoLayerModel method), 58
run_scenarios() (openscm_twolayermodel.base.TwoLayerVariant method), 51
run_scenarios() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel method), 54
run_scenarios() (openscm_twolayermodel.two_layer_model.TwoLayerModel method), 58

S
set_drivers() (openscm_twolayermodel.base.Model method), 51
set_drivers() (openscm_twolayermodel.impulse_response_model.ImpulseResponseModel method), 54

T
TwoLayerModel (class in openscm_twolayermodel.two_layer_model), 57
TwoLayerVariant (class in openscm_twolayermodel.base), 51

U
UnitError, 63

W
with_traceback() (openscm_twolayermodel.errors.ModelStateError method), 63
with_traceback() (openscm_twolayermodel.errors.UnitError method), 63