Documentation for forcing.py

1 Introduction

Radiative Forcing is the difference between sunlight absorbed by the Earth and energy radiated back to space. Positive radiative forcing means Earth receives more incoming energy from sunlight than it radiates to space. This net gain of energy will cause warming. Conversely, negative radiative forcing means that Earth loses more energy to space than it receives from the sun, which produces cooling. This is to say, positive forcing warms the system, while negative forcing cools it. Causes of radiative forcing include changes in insolation and the concentrations of radiatively active gases, commonly known as greenhouse gases and aerosols. This is why we include forcing as a part of our damage model.

According to Wikipedia, radiative forcing can be used to estimate a subsequent change in equilibrium surface temperature ΔT_t arising from that forcing via the equation

$$\Delta T_t = \lambda \Delta F \tag{1}$$

where λ is the climate sensitivity and ΔF is the radiative forcing.

In our case, we take radiative forcing into consideration to better model the damage function. Basically, we assume that the 3 base scenarios have constant mitigation levels RM and calculate their cumulative radiative forcings CRF_t for reference. Given any path with mitigation level x_t's, we calculate cumulative radiative forcings CRF_t and the realized mitigation level RM_t. From RM_t we would like to infer the consequent damage with the $D_t(CRF_t)$ function that we interpolated from the base scenrio simulations. To serve this end, forcing.py is essentially a class that computes the cumulative radiative forcing CRF_t and GHG level GHG_t for a node with mitigation path x_t up to period t in the EZ-Climate model.

2 Model of CRF_t as a Function of x_t

As a cumulative variable, CRF_t depends on the mitigation path up to point t in time. The cumulative mitigation is given by:

$$X_t = \frac{\sum_{s=0}^{t} g_s \cdot x_s}{\sum_{s=0}^{t} g_s} \tag{2}$$

where g_s is the flow of GHG emissions into the atmosphere in period s, for each period up to t, absent any mitigation. The cumulative GHG emissions that must be absorbed into the atmosphere or oceans is $G_t(1 - X_t)$, where $G_t = \sum_{s=0}^{t} g_s$ denotes the cumulative emissions under the BAU scenario.

The radiative forcing is assumed to be given by a log-function fitted to the 3 base case scenarios, named Representative Concentration Pathway (RCP) scenarios in the paper. The

carbon absorption itself is similarly fit to the RCP scenarios, and is assumed to be proportional to the difference between the GHG level in the atmosphere and the cumulative carbon absorption up to that point in time, raise to a power.

According to the paper, radiative forcing in a ten-year interval is given by

$$5.351 \cdot [log(\text{GHG}) - log(278.063)] \tag{3}$$

where GHG is the average level of atmospheric CO_2. The carbon absorption in a ten-year interval is given by

$$0.94835 \cdot \Big|\text{GHG} - (285.6268 + 0.88414 \cdot \sum absorption)\Big|^{0.741547} \tag{4}$$

where the sum is over absorption in previous periods.

The paper does not explain enough about the way forcing.py calculates the variables, but we can formulate the process without understanding the parameters. In particular, forcing.py uses **bau.py** to get GHG emissions under the business-as-usual scenario $\bar{g}_t$ for $t \in \{0, 1, 2, 3, 4, 5, 6\}$. We look into each period of the path in concern by unit called **subinterval** and compare the variables with those of the business-as-usual scenario. Then we calculate **forcing** and **absorbing**, which determine CRF_t and GHG level GHG_t. Their beginning values are:

$$\begin{cases} CRF_0 &= 4.926 \\ GHG_0 &= 400 \\ sink_0 &= 35.396 \end{cases} \tag{5}$$

Here is how we calculate the variables for each subinterval:

- For each period t:

$$\begin{cases} \text{beginning emission} & g_{t,0} = (1 - x_t) \times \bar{g}_t \\ \text{ending emission} & g_{t,t} = \begin{cases} (1 - x_t) \times \bar{g}_{t+1}, & \text{if } t < 5 \\ (1 - x_t) \times \bar{g}_t, & \text{else} \end{cases} \end{cases} \tag{6}$$

- For each subinterval i:

$$\begin{cases} \text{p\_co2}_i &= 0.71 \times \left[g_{t,0} + i \times \frac{g_{t,t} - g_{t,0}}{\text{number of subintervals in period } t}\right] \\ \text{p\_c}_i &= \frac{\text{p\_co2}_i}{3.67} \\ \text{add\_p\_ppm}_i &= \text{length of subinterval} \times \frac{\text{p\_c}_i}{2.13} \\ \text{lsc}_i &= 285.6268 + \text{cum\_sink}_i \times 0.88414 \\ \text{absorption}_i &= 0.5 \times 0.94835 \times |GHG_i - \text{lsc}_i|^{0.741547} \\ \text{cum\_sink}_i &= \text{cum\_sink}_{i-1} + \text{absorption}_i \\ GHG_i &= GHG_{i-1} + \text{add\_p\_ppm}_i - \text{absorption}_i \\ CRF_i &= CRF_{i-1} + 0.13183 \times |GHG_i - 315.3785|^{0.607773} \end{cases}$$

3 Inputs

The methods embedded in the Forcing class require similar inputs that are explained below.

- **m**: an array of fractional mitigation levels that are denoted x_t in the paper.
- **node**: an integer that represents the node in the **TreeModel** for which forcing is to be calculated.
- **tree**: the **TreeModel** object whose tree structure is to be used.
- **bau**: the **BusinessAsUsual** object that gives the emission levels under the business-as-usual scenario.
- **subinterval_len**: a float that represents the length of a subinterval. It is the size of intervals we model the probability of hitting a Tipping Point in the damage function. Within one period, the subintervals of this length are call **increments**. In our example, this value is 5.
- **returning**: an optional string that selects the output. Valid returns include "forcing", "ghg", and "both".
  - "forcing": returns the forcing only;
  - "ghg": returns the GHGs level only;
  - "both": returns both the GHGs level and forcing.

4 Python: Forcing

```
from __future__ import division
import numpy as np
```

4.1 Attributes

The attributes of the Forcing class are basically parameters of the theoretical model, so they are all of the float type. The meanings of those in red are still unknown. See Equation 4 for reference.

- sink_start = 35.596
- forcing_start = 4.926
- forcing_p1 = 0.13173
- forcing_p2 = 0.607773
- forcing_p3 = 315.3785

- **absorption_p1** = 0.94835
- **absorption_p2** = 0.741547
- **lsc_p1** = 285.6268
- **lsc_p2** = 0.88414

```
class Forcing(object):
        """Radiative forcing for the EZ-Climate model. Determines the excess energy cr
        by GHGs in the atmosphere.

        Attributes
        ----------
        sink_start : float
                sinking constant
        forcing_start : float
                forcing start constant
        forcing_p1 : float
                forcing constant
        forcing_p2 : float
                forcing constant
        forcing_p3 : float
                forcing constant
        absorption_p1 : float
                absorption constant
        absorption_p2 : float
                absorption constant
        lsc_p1 : float
                class constant
        lsc_p2 : float
                class constant

        """

        # parameters that I have no idea about
        sink_start = 35.596
        forcing_start = 4.926
        forcing_p1 = 0.13173
        forcing_p2 = 0.607773
        forcing_p3 = 315.3785
        absorption_p1 = 0.94835
        absorption_p2 = 0.741547
        lsc_p1 = 285.6268
        lsc_p2 = 0.88414
```

4.2 Methods

There are 3 methods in this file, while **forcing_at_node** and **ghg_level_at_node** simply executes **forcing_and_ghg_at_node** upon different requests. **forcing_and_ghg_at_node** implements the computation along a deterministic path and gives outputs specified by **forcing_at_node** and **ghg_level_at_node**.

forcing_and_ghg_at_node: calculates the radiative forcing based on GHGs evolution that leads up to damage calculation. The steps are given below.

```
@classmethod
def forcing_and_ghg_at_node(cls, m, node, tree, bau, subinterval_len, returning=
        """Calculates the radiative forcing based on GHG evolution leading up
        damage calculation in `node`.

        Parameters
        ----------
        m : ndarray
                array of mitigations
        node : int
                node for which forcing is to be calculated
        tree : `TreeModel` object
                tree structure used
        bau : `BusinessAsUsual` object
                business-as-usual scenario of emissions
        subinterval_len : float
                subinterval length
        returning : string, optional
                * "forcing": implies only the forcing is returned
                * "ghg": implies only the GHG level is returned
                * "both": implies both the forcing and GHG level is returned

    Returns
    -------
    tuple or float
        if `returning` is
                * "forcing": only the forcing is returned
                * "ghg": only the GHG level is returned
                * "both": both the forcing and GHG level is returned

        """
```

- Specify the case when the node is 0.

```
            #for the start state, return 0 for forcing and ghg_start for the g
            #call bau to get the ghg level
```

```
if node == 0:
        if returning == "forcing":
                return 0.0
        elif returning== "ghg":
                return bau.ghg_start
        else:
                return 0.0, bau.ghg_start
```

- Based on the node given, find its period, path, and decision times through **TreeModel**.
- Determine the number of increments within each period, of the length specified by **subinterval_len**. (This number is converted into integer later, for convenience of computation.)

```
# get the period and the path that the target node are in
period = tree.get_period(node)
path = tree.get_path(node, period)
# the decision time is the time when we make a mitigation, i.e. ar
period_lengths = tree.decision_times[1:period+1] - tree.decision_ti
#increments are the number counts of subintervals within a period
increments = period_lengths/subinterval_len
```

- Assign the starting values of forcing and GHGs level.

```
#assign beginning values
cum_sink = cls.sink_start
cum_forcing = cls.forcing_start
ghg_level = bau.ghg_start
```

- For each period that a node has undergone, determine its beginning and ending GHG levels g_t under mitigation with the base case outputs from **bau.emission_by_decisions**. We assume that the emission level remains constant since the second to last period.

```
for p in range(0, period):
        #for each period, we calculate the start_emission and end_
        #! problem: when will the act takes in to effect? either u
        #emission_by_decision: the emission level at a decision po
        start_emission = (1.0 - m[path[p]]) * bau.emission_by_decis
        if p < tree.num_periods-1: #if not too late to implement m
                end_emission = (1.0 - m[path[p]]) * bau.emission_by
        else:
                end_emission = start_emission #emission level remai
        increment = int(increments[p])
```

- Within each period that a node has undergone, interpolate linearly the emission levels at the beginnings of all the increments in concern. Based on the emission levels, calculate the amounts of consequent ppm, absorption, and forcing.
- Calculate the GHG level at the beginning of each increment by updating values for

forcing and absorption based on the GHG emissions.

- **p_co2** precise meanings unknown
- **p_c**
- **add_p_ppm**
- **absorption**: $0.5 \times 0.94835 \left| GHG - (285.6268 + 0.88414 \cdot \sum absorption) \right|^{0.741547}$ This is the carbon absorption in a 5-year interval. Since our subinterval is of length 5, the absorption should be half of that in Equation 4.
- **cum_forcing**: the cumulative forcing is given by $0.13173 \times |GHG - 315.3785|^{0.607773}$

```
# for each increment in a period, the forcing is affecting
for i in range(0, increment):
        #allocate the emission level change across time
        p_co2_emission = start_emission + i * (end_emission
        p_co2 = 0.71 * p_co2_emission
        p_c = p_co2 / 3.67
        add_p_ppm = subinterval_len * p_c / 2.13
        lsc = cls.lsc_p1 + cls.lsc_p2 * cum_sink
        absorption = 0.5 * cls.absorption_p1 * np.sign(ghg_
        cum_sink += absorption
        cum_forcing += cls.forcing_p1*np.sign(ghg_level-cls
        ghg_level += add_p_ppm - absorption
```

- Return the outcome according to the selection.

```
if returning == "forcing":
        return cum_forcing
elif returning == "ghg":
        return ghg_level
else:
        return cum_forcing, ghg_level
```

forcing_at_node: returns the cumulative radiative forcing CRF_t at a given node that leads up to the damage calculation.

```
@classmethod
def forcing_at_node(cls, m, node, tree, bau, subinterval_len):
        """Calculates the forcing based mitigation leading up to the
        damage calculation in `node`.

        Parameters
        ----------
        m : ndarray
                array of mitigations in each node.
        node : int
```

```
            the node for which the forcing is being calculated.

        Returns
        -------
        float
            forcing

        """

        return cls.forcing_and_ghg_at_node(m, node, tree, bau, subinterval_len,
```

ghg_level_at_node: returns the GHGs level at a given node that leads up to the damage calculation.

```
    @classmethod
    def ghg_level_at_node(cls, m, node, tree, bau, subinterval_len):
        """Calculates the GHG level leading up to the damage calculation in `n

        Parameters
        ----------
        m : ndarray
            array of mitigations in each node.
        node : int
            the node for which the GHG level is being calculated.

        Returns
        -------
        float
            GHG level at node

        """
        return cls.forcing_and_ghg_at_node(m, node,tree, bau, subinterval_len, r
```