Documentation for forcing.py

1 Introduction

forcing.py is essentially a class that computes the radiative forcing for the EZ-Climate model. It helps determine the excess energy created by GHGs in the atmosphere.

2 Model of Excess Energy from GHGs

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.
- **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.

- **sink_start**: Its default value is 35.596.
- **forcing_start**: Its default value is 4.926.
- **forcing_p1**: Its default value is 0.13173.
- **forcing_p2**: Its default value is 0.607773.
- **forcing_p3**: Its default value is 315.3785.
- **absorption_p1**: Its default value is 0.94835.
- **absorption_p2**: Its default value is 0.741547.
- **lsc_p1**: Its default value is 285.6268.
- **lsc_p2**: Its default value is 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

forcing_and_ghg_at_node: calculates the radiative forcing based on GHGs evolution that leads up to damage calculation.

```
@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 t
        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 measured at the length specified by **subinterval_len**.

```
# 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. an
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 emission level through **bau.emission_by_decisions**. We assume that the emission level remains constant since the second to last period.
- For each period that a node has undergone, round its number of increments to an integer.

```
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 w
        #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 rema
        increment = int(increments[p])
```

- Within each period that a node has undergone, allocate the emission level change across time, and find corresponding ...

– **p_co2**

– **p_c**

– **add_p_ppm**

– **absorption**

– **cum_forcing**

```
				# 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 forcing based mitigation 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
```