Documentation for bau.py

1 Introduction

For comparison purpose, it is important for us to generate a baseline, or an emission scenario without mitigation, for reference. This is what we call the business-as-usual scenario, and what we want to realize with the bau.py file. Basically, we assume constant consumption growth and GHG emissions grow over time without mitigation. Notice that the paper does not take damage into account for the constant growth of consumption.It may be the case that people can produce more to maintain a constant consumption growth, despite the damage from GHG emissions. The bau.py has two classes: **BusinessAsUsual** and **DLWBusinessAsUsual**, where **BusinessAsUsual** is the superclass of **DLWBusinessAsUsual**. BusinessAsUsual is designed for the users to define other business-as-usual assumptions. They can create new business-as-usual model like non-linear growth of GHG emission by writing their own class that inherits the base class BusinessAsUsual.

- **BusinessAsUsual**: an abstract class that provides the general structure for the emission growth model under business-as-usual scenario. This is the superclass that specifies necessary variables and returns GHG emissions (for instance. Users who use the EZ-Climate python project can take their own business-as-usual assumptions and create new BAU classes that inherit the base class BusinessAsUsual.
- **DLWBusinessAsUsual**: a class that models the DLW emission model under business-as-usual scenario. It returns the BAU GHG emission level at a given time point, which we refer to a lot when we implement other objects, like cost.py and damage.py. Indeed, DLWBusinessAsUsual is a child of BusinessAsUsual, and it inherits the variables and functions from BusinessAsUsual to start with.

Since this file contains 2 classes, we discuss them separately.

1.1 Emission Level, GHG Level, and Emission Amount

For the business-as-usual scenario we mainly deal with 3 concepts regarding GHG: emission level, GHG level, and emission amount.

- At each time point, certain amount of GHG emission takes place. We call the GHG emission at this time point **GHG emission level**
- **GHG level** refers to the greenhouse gas concentration
- we also need to work with the flow of GHG emissions into the atmosphere in each period t up to T absent of any mitigation, we call it g_t.

According to DLW paper, we extrapolate the results from Wagner and Weitzman and assume the GHG level will increase to 1000 (ppm of CO_2) if no mitigation is to be done. Since our damage function is based on the maximum GHG level in the last period, we would like to

first calculate the GHG levels throughout the whole time span. To do this, we take in inputs of emission levels at some future points. We assume that the emission level changes smoothly with time and the GHG level change has a linear relationship with the GHG emission amount for each period, g_t. That is, for a period with a relatively large amount of GHG emission each unit of time, it is likely that this period experiences greater change in GHG level.

2 BusinessAsUsual

2.1 Inputs

- **ghg_start**: a float that represents GHG level in ppm in the first period. In our example, this value is 400.0.
- **ghg_end**: a float that represents the projected GHGs level in ppm in the last period. In our example, this value is 1000.0.

2.2 Python

Import necessary packages.

```
import numpy as np
from abc import ABCMeta, abstractmethod

class BusinessAsUsual(object):
    """Abstract BAU class for the EZ-Climate model.

    Parameters
    ----------
    ghg_start : float
        today's GHG-level
    ghg_end : float
        GHG-level in the last period

    Attributes
    ----------
    ghg_start : float
        today's GHG-level
    ghg_end : float
        GHG-level in the last period
    emission_by_decisions : ndarray
        emission levels at decision points
    emission_per_period : ndarray
        amounts of emission for each single period.
```

```
    emission_to_ghg : ndarray
        change in GHGs level attributed to each period
    emission_to_bau : float
        constant: the last period increase in GHGs level divided by its emission

    """
__metaclass__ = ABCMeta
    def __init__(self, ghg_start, ghg_end):
        self.ghg_start = ghg_start
        self.ghg_end = ghg_end
        self.emission_by_decisions = None
        self.emission_per_period = None
        self.emission_to_ghg = None
        self.emission_to_bau = None
        self.bau_path = None

   @abstractmethod
    def emission_by_time(self):
        pass
```

3 DLWBusinessAsUsual

3.1 Inputs

DLWBusinessAsUsual requires inputs about the green house gases (GHGs) and emissions.

- **ghg_start**: a float that represents current GHGs level (the first period). In our example, this value is 400.0.
- **ghg_end**: a float that represents ending GHG level (the last period). In our example, this value is 1000.0.
- **emit_time**: a list or array that represents time points, in years, from now when emissions occur. In our example, emit_time=[0, 30, 60], so we are given emission levels 0, 30, 60 years later. Based on emission information available at **emit_time**, we can infer emission amounts and GHG levels through the whole time span. Notice that the first entry stands for the start point of our interpolation, so we call it the **start point** in this model. The paper says nothing about why the emission intervals are of 30 years.
- **emit_level**: a list or array that represents given emission levels at **emit_time**. In our model, emit_level = [52, 70, 81.4] The paper does not mention where these numbers come from.

3.2 Attributes

DLWBusinessAsUsual has some attributes that reveal the relationship between projected GHGs levels and emission, including some that we've seen in **Inputs**.

- **ghg_start**: a float that represents current GHG level.
- **ghg_end**: a float that represents GHG level in the last period.
- **emit_time**: a list or array that represents time points, in years, from now when emission levels are given. See **Inputs** for explanation in detail.
- **emit_level**: a list or array that represents emission levels at **emit_time**.
- **emission_by_decision**: an array that represents the emission levels at different decision times.
- **emission_per_period**: an array that represents the amount of emission during an arbitrary period.
- **emission_to_ghg**: an array that represents the amounts of change in GHG level attributed to each period according to the weights of emission amount during the single period **emission_per_period** over the total amount of emission through the whole time span.
- **emission_to_bau**: a float that represents the change of GHGs level attributed to the last period divided by the amount of emission during the last period. We call it the **bau_factor** in the formula to follow.

3.3 Python

Define the DLWBusinessAsUsual with default input values as well as values inherited from the super class BusinessAsUsual.

```
class DLWBusinessAsUsual(BusinessAsUsual):
    """Business-as-usual scenario of emissions. Emissions growth is assumed to slow do
    exogenously - these assumptions represent an attempt to model emissions growth in
    business-as-usual scenario that is in the absence of incentives.

    Parameters
    ----------
    ghg_start : float
        today's GHG-level
    ghg_end : float
        GHG-level in the last period
    emit_time : ndarray or list
        time, in years, from now when emissions occur
    emit_level : ndarray or list
```

```
        emission levels in future times `emit_time`

    Attributes
    ----------
    ghg_start : float
        today's GHG-level
    ghg_end : float
        GHG-level in the last period
    emission_by_decisions : ndarray
        emission level a  specific decision time
    emission_per_period : ndarray
        the amount of emission for each period/between two decision times
    emission_to_ghg : ndarray
        change in GHGs level attributed to each period
    emission_to_bau : float
        constant: the last period increase in GHGs level divided by its emission
    emit_time : ndarray or list
        time, in years, from now when emissions occurs
    emit_level : ndarray or list
        emission levels in future times `emit_time`

    """

    #the default emit_time [0, 30, 60] should  work here, but with a tree model with d
    #[2015, 2030, ...]
    def __init__(self, ghg_start=400.0, ghg_end=1000.0, emit_time=[0, 30, 60], emit_leve
        super(DLWBusinessAsUsual, self).__init__(ghg_start, ghg_end)
        self.emit_time = emit_time
        self.emit_level = emit_level
```

3.3.1 Methods

emission_by_time: infers the emission level at a given time point, since the start point. The code implements the calculation in the following way:

- Take an integer input that represents a future time point in years. For example, an input 30 means 30 years from the start point.
- Assume a linear relationship between time and emission level. Let the given emit levels **emit_level** at specified emit times **emit_time** t_0, t_1, t_2 denoted by $\text{emit}_0, \text{emit}_1, \text{emit}_2$. Then infer the emission level based on the time.

  In general, emission by time $t = \begin{cases} \text{emit}_0 + t \cdot \frac{\text{emit}_1 - \text{emit}_0}{t_1 - t_0}, & 0 \le t < t_1. \\ \text{emit}_1 + (t - t_1) \cdot \frac{\text{emit}_2 - \text{emit}_1}{t_2 - t_1}, & t_1 \le t < t_2 \\ \text{emit}_2, & t_2 \le t \end{cases}$

- Return the emission level.

```
def emission_by_time(self, time):
    """Returns the BAU emissions at any time

    Parameters
    ----------
    time : int
        future time period in years

    Returns
    -------
    float
        emission

    """
    if time < self.emit_time[1]:
        emissions = self.emit_level[0] + float(time) / (self.emit_time[1] - self.emi
                    * (self.emit_level[1] - self.emit_level[0])
    elif time < self.emit_time[2]:
        emissions = self.emit_level[1] + float(time - self.emit_time[1]) / (self.emi
                    - self.emit_time[1]) * (self.emit_level[2] - self.emit_level[1])
    else:
        emissions = self.emit_level[2]
    return emissions
```

bau_emission_setup: interpolates the GHG levels throughout the entire time span. We derive the GHG level path with the following steps:

- Get necessary information about the periods and decision times from tree. Determine the lengths of each period **period_len** between decision times.
- Set **ghg_start** as the start point of the GHG level path **bau_path** under business-as-usual scenario.

```
def bau_emissions_setup(self, tree):
    """Create default business as usual emissions path. The emission rate in e
    assumed to be the average of the emissions at the beginning and at the en

    Parameters
    ----------
    tree : `TreeModel` object
        provides the tree structure used

    """
    num_periods = tree.num_periods
    self.emission_by_decisions = np.zeros(num_periods)
```

```
self.emission_per_period = np.zeros(num_periods)
self.bau_path = np.zeros(num_periods)
self.bau_path[0] = self.ghg_start
self.emission_by_decisions[0] = self.emission_by_time(tree.decision_times[0]
period_len = tree.decision_times[1:] - tree.decision_times[:-1]
```

- Calculate **emission_by_decisions**, the emission level at each decision points.
- Take the average emission rate of a period to be the average of the emission levels at the beginning and the end of the period. Calculate the amount of emission for each single period, g_t, by multiplying its average emission rate with the length of the period. **emission_per_period** consists of g_t's for all the periods defined by decision times (in the TreeModel):

$$g_t = \text{length of period } t \times \frac{\text{emission levels at decision point } t \text{ and } t+1}{2}$$

```
for n in range(1, num_periods):
    self.emission_by_decisions[n] = self.emission_by_time(tree.decision_tim
    self.emission_per_period[n] = period_len[n] * (self.emission_by_decisio
    #the average is the average of the emission level at the beginning and
```

- Finally, we need to allocate the assumed GHG level change across time. In particular, we attribute the total change $1000 - 400 = 600$ according to the weight of the emission amount in each period $\frac{g_t}{\sum_{t \in periods} g_t}$:

  Translate the emissions to corresponding GHG levels:

  - sum of emission $= \sum_{t \in \text{period}} g_t$
  - ΔGHG = total change in GHG level = **ghg_end** − **ghg_start**
  - $\Delta\text{GHG}_t = \Delta\text{GHG} \times \frac{g_t}{\text{sum of emission}}$, gives the attributed change in GHG level for each period, **emission_to_ghg**.

    ```
    #the total increase in ghg level of 600 (from 400 to 1000) in the bau
    self.emission_to_ghg = (self.ghg_end - self.ghg_start) * self.emission_
    ```

  - bau factor $= \frac{\Delta\text{GHG}_T}{g_T}$, gives the ratio of change in GHG and emission amount during the last period, **emission_to_bau**.

    ```
    #emission_to_bau is essentially the last period ghg level divided by
    self.emission_to_bau = self.emission_to_ghg[-1] / self.emission_per_per
    ```

  - Get a complete path **bau_path** with entries being GHG level at each decision point t: $\text{GHG}_t = \text{GHG}_{t-1} + g_t \times$ bau factor

    ```
    #find the bau_path based on the last period's emission_to_bau
    for n in range(1, num_periods):
        self.bau_path[n] = self.bau_path[n-1] + self.emission_per_period[n]
    ```

actually i do not know why it calculate bau path in this way (using bau factor). why not use emission_to_ghg