Documentation for damage.py

1 Damage Function Specification

The climate change function $D_t(CRF_t, \theta_t)$ captures the fraction of endowed consumption that is lost due to damages from climate change. If $D_t(CRF_t, \theta_t) = 0$, the agent would receive the full consumption endowment. However, damage from climate change can push D_t above zero. D_t, in turn, depends on 2 variables: CRF_t, which we define as the cumulative solar radiative forcing up to time t, which determines global average temperature, and θ_t, the Earth's fragility, a parameter that characterizes the uncertain relation between the global average temperature and consumption damages. (See more explanation of forcing in forcing.py's documentation.) We assume fragility θ_t to evolve stochastically and will explain more in the subsection to follow.

We compare CRF_t to 3 baseline emissions paths, g_t, for which we have created associated damage simulations via **damage_simulation.py**. The one way, then, to affect the level of damages is to change mitigation across time, x_t. In terms of the damage, its specification has two components: a non-catastrophic component and a catastrophic one, triggered by crossing a particular threshold. If the threshold is hit at any time, we assume the catastrophic damage has a negative effect on consumption in all future periods.

We calculate the overall damage function $D_t(CRF_t, \theta_t)$ for the baseline emissions paths, g_t, using Monte-Carlo simulation. As we described in detail below, we run a set of simulations for each of three constant mitigation levels implied in the base scenarios, which determine cumulative radiative forcing at each point in time. In each run of the simulation, we draw a set of random variables:

- global average temperature change;
- the parameter characterizing non-catastrophic damages as a function of temperature;
- an indicator variable that determines whether or not the atmosphere hits a tipping point at any particular time and state;
- the tipping point damage paramter.

With the simulation results, we use **damage.py** to incorporate other variables like CRF_t (via **forcing.py**) to determine the damage coefficients and interpolate damage function for different mitigation/forcing levels. According to the paper, the **state variable** θ_t stands for the fragility and indexes the distribution results from these sets of simulations. Interpolation across the three mitigation levels gives us a continuous function D_t across cumulative radiative forcing levels CRF_t. Details about θ_t will come in the sections to follow.

1.1 The Specification of Temperature as a Function of GHG Levels

The distribution of temperature changes as a function of mitigation strategies is calibrated to 3 base scenarios, signified by the maximum level of CO_2 in the atmosphere. We take the calibration of Wagner and Weitzman (2015) and set the zero-mitigation scenario by extrapolating the CO_2-equivalent concentration to 1000 ppm. Then we assume that 100% mitigation will lead to a maximum GHG level of 400 ppm. Other levels of average mitigation are assumed to result in damages associated with GHG levels linearly interpolated between 400 ppm and 1000 ppm. Thus, an average mitigation of 50% over time is said to give an interpolated damage associated with a maximum GHG level of 700 ppm at the final period.

Table 1 gives the probability of different levels of ΔT_{100} - the temperature change over the next 100 years - for given maximum levels of GHG in atmosphere. $T \in \{2, 3, 4, 5, 6\}$ is in Celsius. For example, the probability that the temperature will change by 3°C in 100 years, given that the maximum GHG level will be 450 ppm by that time, is 0.13.

Table 1: Prob($\Delta T_{100} > T$) for 3 Specified Mitigation Levels

T	450	650	1000
2 °C	0.40	0.85	0.99
3 °C	0.13	0.54	0.86
4 °C	0.04	0.30	0.66
5 °C	0.02	0.15	0.46
6 °C	0.00	0.07	0.30

We then use assumptions akin to Pindyck (2012) to fit a displaced gamma distribution around final GHG concentrations (**ghg_levels**), while setting levels of GHG 100 years in the future equal to equilibrium levels. The formula is given by:

$$f(x; \alpha, \beta, \theta) = \frac{(x+\theta)^{\alpha-1} e^{-\frac{x+\theta}{\beta}}}{\beta^{\alpha}\Gamma(\alpha)}, x \geq -\theta$$

where $\Gamma(\alpha) = \int_0^{\infty} s^{\alpha-1} e^{-s} ds$ is the Gamma function.

Table 2 gives the parameters for these distributions, and the probabilities from the fitted displaced gamma distributions, which line up well with the numbers in Table 1.

Table 2: Fitted Values of Prob($\Delta T_{100} > T$) for 3 Specified Mitigation Levels

T	450	650	1000
2 °C	0.40	0.87	0.99
3 °C	0.14	0.57	0.91
4 °C	0.04	0.29	0.70
5 °C	0.01	0.12	0.44
6 °C	0.00	0.05	0.24
Alpha	2.810	4.630	6.100
Beta	0.600	0.630	0.670
Theta	-0.25	-0.5	-0.9

We use **_pindyck_simulation** in damage_simulation.py to realize the simulation of temperature change for the 3 base scenarios. theta_1000 is -1.0 instead of -0.9 in the code. To obtain the temperature distribution at other times, we follow Pindyck (2012) and specify that the time path for the temperature change at time t (in years) is given by:

$$\Delta T(t) = 2\Delta T_{100}\left[1 - 0.5^{\frac{t}{100}}\right] \tag{1}$$

The **_interpolation_of_temp** method in damage_simulation.py takes the cumulative temperature change up to the final period, ΔT_{100}, and gives the corresponding temperature changes at each decision points, $\Delta T(t)$. As time increases, the temperature change asymptotes to double the value of ΔT_{100}. (This is how we define **maxh** in damage_simulation.py: it represents the time it takes to achieve half the maximum temperature change given the GHG scenario, so it equals 100 in our case) We would like to emphasize that both the distribution of ΔT_{100} and the functional form for the path in equation 1 merit further scientific scrutiny.

1.2 The Specification of Damage as a Function of Temperature

Based on the temperature change, we use damage functions to convert the temperature change into global mean economic losses. As mentioned earlier, our damage function has both a catastrophic component and a non-catastrophic one. However, the functional form for each damage function component contains a parameter that characterizes the uncertainty in our present understanding of this relationship. Our code adopts the EZ-Climate model where the agent knows the form of the distribution of this parameter at the initial date, and in each period she learns more about the distribution of the parameter. However, the final realization of the parameter is unknown until the next-to-last period. (More in Section 1.3)

The non-catastrophic component of our damages is based on Pindyck (2012), who fits a functional form to data from the IPCC's Fourth Assessment Report, and obtains a loss function of the form:

$$L(\Delta T_t) = e^{-13.97\gamma\Delta T(t)^2} \tag{2}$$

where γ is drawn from a displaced gamma distribution with parameters $r = 4.5, \lambda = 21341$, and $\theta = -0.0000746$. This is what we do with **_pindyck_impact_simulation** in damage_simulation.py, based on the simulated temperature changes in hand.

Based on non-catastrophic damages, consumption in any time t is reduced as follows:

$$CD_t = \bar{c}_t L(\Delta T(t)) \tag{3}$$

We augment Pindyck's damage function with the possibility of catastrophic events after reaching a particular temperature threshold, namely **Tipping Point**, which creates the potential for a much larger impact on consumption. In our specification, **Prob(TP)** denotes the probability of hitting a Tipping Point over a typical interval of fixed length **period** as a function of the global temperature changes as of that time, ΔT_t, and of a parameter, **peakT**:

$$Prob(TP) = 1 - \left(1 - \left[\frac{\Delta T(t)}{max\{\Delta T(t), peakT\}}\right]^2\right)^{\frac{period}{30}} \tag{4}$$

As peakT increases, the probability of reaching a climatic tipping point decreases for a given $\Delta T(t)$. The new paper does not assign a default value but the example takes peakT = 6 and the code sets peakT = 9.

In our simulations, in each period p and for each state, there is a probability $Prob(TP)$ that a tipping point is hit, given $\Delta T(t)$ and peakT. Conditional on the case that a tipping point is hit at time t^* in a simulated path, the level of consumption for each period $t \geq t^*$ is then given by:

$$CDTP_t = CD_t \cdot e^{-TP_{damage}} = \bar{c}_t \cdot L(\Delta T(t)) e^{-TP_{damage}}, \text{ for } t \geq t^* \tag{5}$$

where TP_{damage} is a random variable drawn from a gamma distribution with parameters $\alpha = 1$ and $\beta =$ disaster_tail. The new paper sets disaster_tail = 18. We use **_disaster_cons_simulation** in damage_simulation.py to get appropriate TP_{damage} values.

If we take both the catastrophic and non-catastrophic damages into consideration, the damage function for a given level of mitigation and in a given state of nature is:

$$D_t = (1 - L(\Delta T(t))) \cdot (1 - I_{TP}[1 - e^{-TP_{damage}}]) \tag{6}$$

where I_{TP} is an indicator variable which is equal to one if a tipping point has been hit, and zero otherwise. However, $L(\Delta T(t)), I_{TP}$, and $e^{-TP_{damage}}$ are each dependent on the specific realization of the draws of random numbers in our simulations.

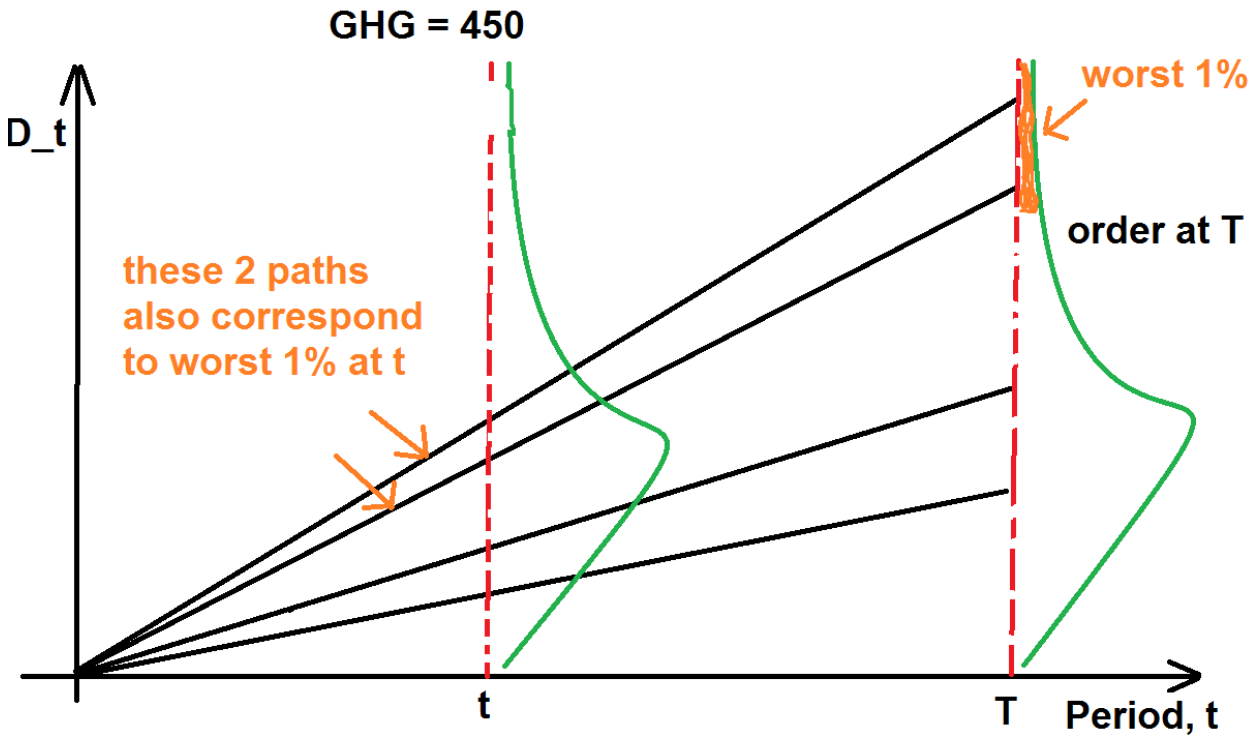
1.3 Damage Function Uncertainty

The mapping from mitigation policy to damages over time, D_t, goes via cumulative radiative forcing, which determines the excess energy created by GHGs in the atmosphere. The damage distribution associated with a given level of radiative forcing is interpolated, or extrapolated, relative to the radiative forcing of damage distributions estimated from the 3 baseline scenarios. The first has eventual atmospheric CO_2 levels of around 1000 ppm. The second assumes constant mitigation leaing to eventual levels of 650 ppm, equivalent to reducing emission by almost 60% relative to the 10000 ppm scenario. The third scenario assumes a constant mitigation of over 90%, leading to eventual CO_2 concentrations of 450 ppm.

For each of the 3 maximum GHG concentration levels - 450, 650, and 1000 ppm - we run a set of random scenarios to generate a distribution of D_t for each period. We order the scenarios based on D_T, the fraction of consumption that is lost due to climate change in the final period. The order applies to interpolated damage D_t in other periods, so the ranking of paths is essentially independent of period t. Notice here that the smaller D_T is, the better the climate change is. For this reason, we order D_T in a descending order as the fragility decreases (see Figure 2) and make it easy to look into the simulated D_T's separately according to their percentiles. To do this, we choose **states of nature**, or the state variable, θ_t with specified probabilities to represent the different percentiles of the D_T results.

The **states of nature**, or state variable, is different from the **states** we define in TreeModel. For the final period, the first state in the final period refers to the first node, but by first state of nature we mean the worst 1% of the D_T's that we get, see Figure 1. To be more specific, as we define the first state of nature to be the worst 1% of the simulated D_t, then we assume the corresponding **damage coefficient** at time t for the given level of mitigation is the average fractional-damage at time t for the smallest 1% values of D_t. More generally, if the k^{th} state of nature represents the simulation outcomes in the range [prob(k-1), prob(k)], then the damage coefficient for the k^{th} state of nature is the average fractional-damage in that range of simulations in which the distribution for D_t lies within those percentiles.

Figure 1: Ordering D_T according to State of Nature



The simulations are used to calculate damages in each period for any particular state of nature, θ_t, and any chosen time path for mitigation actions. We do this by first calculating the radiative forcing associated with each baseline scenario at the end of each period through **_forcing_init**, and then interpolating the damages smoothly between the 3 different simulations with respect to their levels of radiative forcing. Functional forms for both GHG levels and climate forcing as a function of GHG emissions are fitted to the Representative Concentration Pathway (RCP) scenarios adopted by the IPCC for its Fifth Assessment Report (IPCC, 2013). In the IPCC report emissions, GHG concentrations, and radiative forcing are given for each of 3 RCP scenarios. The radiative forcing is assumed to be given by a log-function fitted to these RCP scenarios. 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 the point of time, raised to a power. **forcing.py** does this part for us.

Our task now is to calculate an interpolated damage function using our 3 simulations where we have damage coefficients (for a given state and period) to find a smooth function that gives damages for any particular level of radiative forcing up to each point in time. To do so, we assume a linear interpolation between the 650 and 1000 ppm scenarios, and a quadratic interpolation between 450 and 650 ppm. We find these interpolations with **_damage_interpolation**. In addition, we impose a smooth pasting condition at 650 ppm, having the level and derivative of the interpolation below 650 match the level and slope of the line above.

Below 450 ppm, we assume climate damages exponentially decay toward 0. Mathematically, we let $S = \frac{d \cdot p}{l \cdot ln(0.5)}$, where d is the derivative of the quadratic damage interpolation function at 450 ppm, $p = 0.91667$ is the average mitigation in the 450 simulation, and the level of damage is l. Radiative forcing at any point below 450 ppm then is x percent below that of the 450 ppm simulation, with $x = \frac{R-r}{R}$, where R is the radiative forcing in the 450 ppm simulation and r is the radiative forcing given the mitigation policy. Letting $\sigma = 60$, the extension of the damage function for $x > 0$ is defined as

$$\text{Damage}(x) = l \cdot 0.5^{(x \cdot S)} e^{-[(x \cdot p)^2/\sigma]} \tag{7}$$

———-

Our task is to calculate an interpolated damage function between the 3 scenarios where we have damage coefficients (for a given state and period) to find a smooth function that gives damages for any particular average mitigation percentage up to each point in time. Can we graphically illustrate this interpolation?

We first calculate a quadratic section of the damage function which starts (for a given state and period) at the level of damages in the 1000 ppm maximum GHG scenario and is assumed to have a zero derivative at that point. The curvature as a function of mitigation is calculated such that the damage function matches the damage coefficient at the 650 ppm maximum GHG scenario. For emissions mitigation percentages less than 58.3% we use this quadratic curve to interpolate damages.

We next calculate a quadratic section of the damage function which starts at the level of

damages in the 650 ppm maximum GHG scenario is assumed to have a derivative equal to that of the first quadratic where they meet at the 58.3% emissions mitigation point. The curvature of the second quadratic is then calculated such that the damage function matches the damage coefficient at the 450 ppm maximum GHG scenario. We use this quadratic curve to interpolate damages when emissions mitigation is greater than 58.3% and less than 100%.

We allow for the possibility of net GHG removal from the atmosphere, in which case emissions mitigation can exceed 100%. In that case we extend the second quadratic interpolation but decay it toward 0 by dividing by $2^{10(\%\text{mitigation}-1)}$. Thus at 110% mitigation we divide by 2; at 120% mitigation we divide by 4; etc. The purpose of this decay is to cause the quadratic curve to smoothly decay toward 0 damages.

———-

The climate sensitivity - summarized by state of nature θ_T - is not known prior to the final period. Rather, what the representative agent knows is the distribution of possible final states, θ_T. We specify that the damage in period t, given a cumulative radiative forcing CRF_t up to time t, is the probability weighted average of the interpolated damage function over all final states of nature reachable from that node. Specifically, the damage function at time t, for the node indexed by θ_t is assumed to be

$$D_t(CRF_t, \theta_t) = \sum_{\theta_T} \Pr(\theta_T|\theta_t) \cdot D_t(CRF_t, \theta_T) \tag{8}$$

where the sum is taken over all states that are possible from the node indexed by θ_t (i.e., for which $\Pr(\theta_T|\theta_t) > 0$). This is what we do in **_damage_function_node**, where we sum up probability-weight damage at reachable final states.

1.4 Damages for Concentrations Below Pre-industrial Levels

Introducing carbon dioxide removal (CDR) technologies, combined with stochastic fragility θ_t creates a unique possibility: that, in some states of the world, GHG concentrations may fall below pre-industrial levels of 280 ppm. There is nothing magical about 280 ppm-in an absolute sense, it may not bet the 'optimal' climate to begin with-but it does serve as the baseline for damage calculations based on global warming above pre-industrial levels. It is clear that going (well) below 280 ppm would lead to climate damages, much like going (well) above 280 ppm does. We introduce a simple penalty function of the form:

$$f(x) = [1 + e^{k(x-m)}]^{-1} \tag{9}$$

where m is the level of GHG concentrations where calibrated at half the total penalty and k is a simple scalar. For our base case calibration, we use $m = 200$ and $k = 0.05$. The benefit of a low k and thus a smooth penalty function is largely computational. More importantly, the calibration ensures that the penalty Equation 9 at 280 ppm is close to zero.

We use Equation 7 and 9 to manage the special concentration cases in **_damage_function_node**.

1.5 Tree Structure

Figure 3 illustrates the tree structure employed in EZ-Climate's baseline analysis. Beginning with the first node, in 2015, the agent is assumed to know the structure of the decision tree, the state probabilities, and the damage function in each future state of the world. In the baseline model, where a move up or down in each period is equally likely, the probabilities of the final states are given by a binomial distribution, the simplest possible probability representation.

Another feature evident in Figure 3 is particularly important, given our use of the Epstein-Zin preference specification: the recombining tree structure. This implies 2 features: for one, the damage function in each state (after period 0) is independent of the way in which information was revealed at the end of each period. For example, the damage function in state 'uud' (the blue path in Figure 3) is identical to that in 'udu' (green) and 'duu' (red).

Second, the agent's utility is path-dependent. The history of mitigation depends on the process by which the agent learns the state. Thus, consumption, and mitigation, will depend upon the path. Consequently, in solving for the agent's utility along each of these paths, we need to keep track of the path by which the agent learned about the damage function. Consumption decisions depend on it.

For example, the consumption flow at the start of period 1 is given by $c_1 = \bar{c_0} \cdot e^{0.015 \times 15}(1 - D_1(CRF_1, \theta_1) - \kappa_1(x_1))$. That is, the consumption at the start of period 1, c_1, is equal to endowed consumption minus the fractional cost of damages and of mitigation chosen at the beginning of period 1. Mitigation is optimally chosen by the agent, and is therefore a function of the state - mitigation will be lower if the agent learns that the world is in state d rather than state 'u'.

This analysis gives us consumption levels c_0 and c_1 respectively. To interpolate between c_0 and c_1, we fit an exponential growth function to consumption levels, using 5-year intervals. Note that this is equivalent to assuming that immediately after choosing the mitigation level in period 0, the agent's consumption starts to reflect climate changes from the first revealed state. However, she is not allowed to change the period 0 mitigation to relect this knowledge until the next period.

2 Introduction to Classes

The file damage.py contains 2 classes that are designed for the damage function in DLW's paper: **Damage** and **DLWDamage**. Similar to the bau.py file, Damage is the super class of DLWDamage. While DLWDamage inherits the variables and functions from Damage, DLWDamage is based on the damage model in the paper and designed to find the corresponding outcomes. What's the point in creating a superclass if there is only one class that inherits from it?

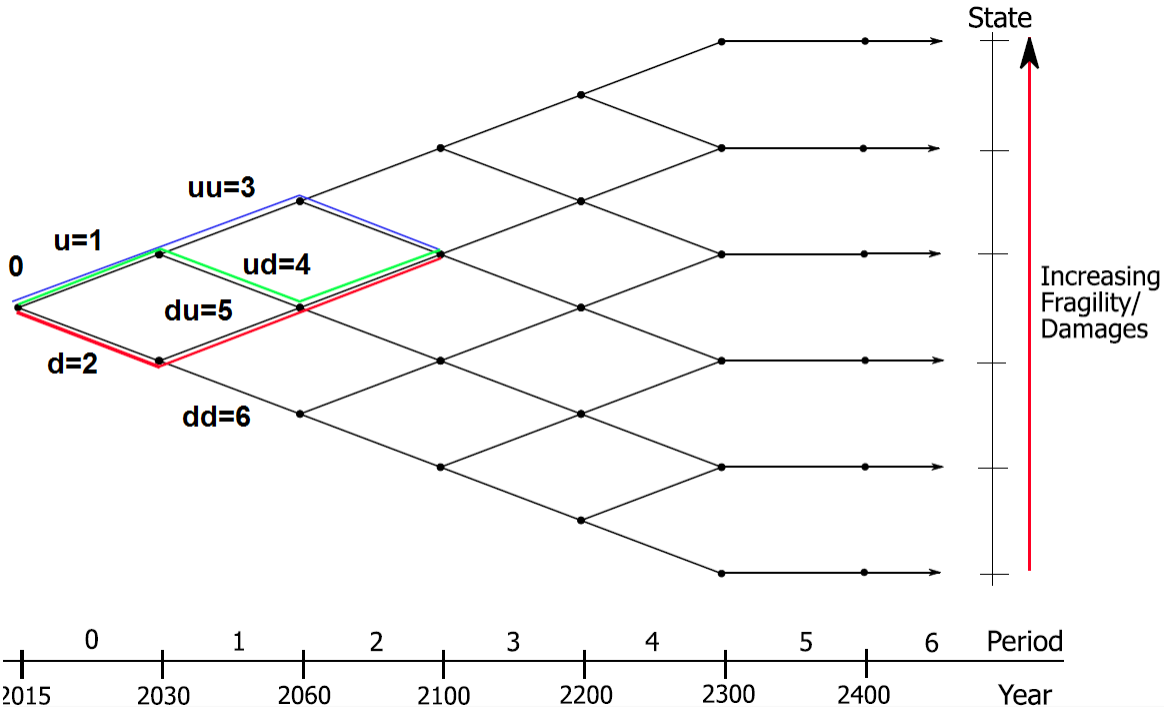
- **Damage**: an abstract damage class for the EZ-Climate model.

- **DLWDamage**: the class that determines the damage under the EZ-Climate model. It provides the damage from emissions as well as mitigation outcomes.

2.1 Recombining Tree v.s. Non-Recombining Tree

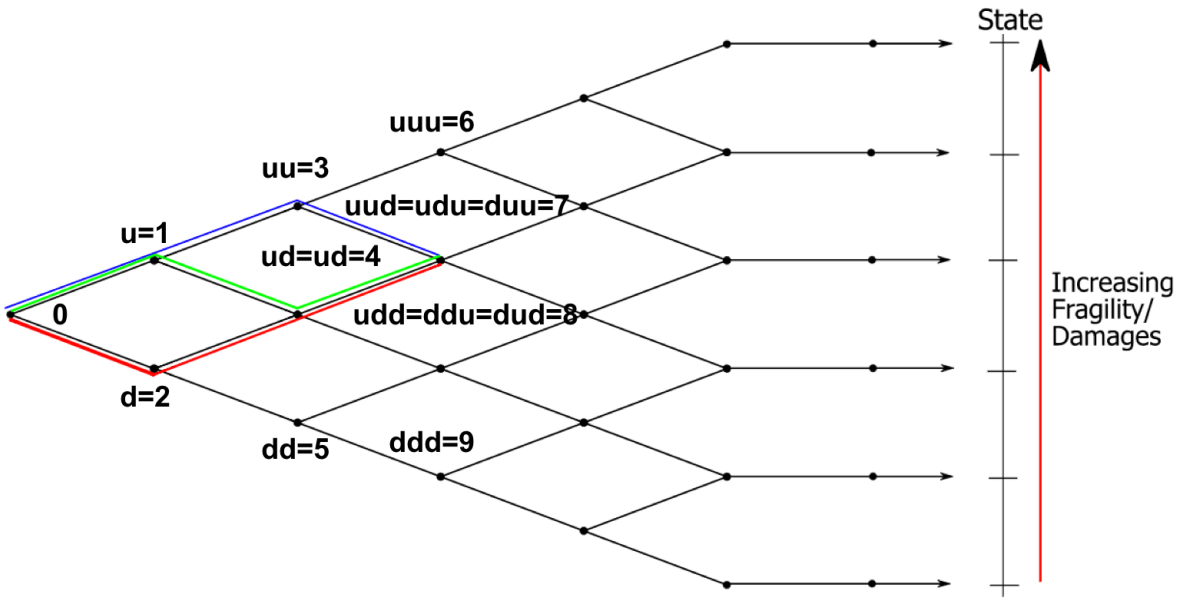
An important concept in our damage calculation is to store transform nodewise information, such as damage, stored in the TreeModel from a **recombining** model to a **nonrecombining** one. These two tree models essentially differ in the relationships between nodes and paths, as the nodes are no longer dependent on the paths in a recombining tree. Figure 2 and 3 give 2 examples of these tree models with 6 periods. In general, an upward movement signals a future movement path that is different from the one a downward movement signals, so the results of an up-down movement usually differ from those of a down-up movement. This is what we typically call a non-recombining tree, where each node independently have two children nodes. The TreeModel in tree.py is a good example and it has 32 states in the last period. In finance, however, another popular model is the recombining tree. Unlike the non-recombining tree, a recombining tree describes asset price movements in which an up-down movement generates the same price outcome as a down-up movement as long as they have the same start and ending points. This model takes 'uud' and 'duu' as the same state and has only 6 states in the last period.

Figure 2: Non-recombining Tree



Although a non-recombining model is more realistic, we can take the advantage of a recombining tree as long as the variables in concern are path-independent. For example, when we simulate D_T in damage_simulation, we do not consider path-dependent variables like

Figure 3: Recombining Tree



mitigation level, so the simulation results in hand are essentially independent of path. In this case, we can treat the model as a recombining one and infer analogous variables in a non-recombining model. This transformation can speed up our computation quite a lot.

In our DLWDamage model, we assume that the simulated D_T results follow a recombining tree. This is to say, under an arbitrary scenario of maximum GHG level, movements along different paths from one state to another always give us the same fragility, θ_t, as long as the paths have the same start and ending points. Therefore, we transform the damage stored in a non-recombining tree into one stored in a recombining tree. For the rest of the project, we stick to the non-recombining tree model for the convenience of data storage and computation.

2.1.1 Transformation

When it comes to the transformation done with **_recombine_nodes**, we need to filter the simulation results and classify them according to the final states under the recombining tree model. By classifying, we want to put simulation results of the same final states in the recombining tree into one class. To realize this transformation, we do not build a new tree model with fewer nodes. Instead, we reorganize the results from our non-recombining tree and infer the recombining outcomes. In order to clarify the process, we need to introduce 3 new terms:

- **old state**: the final states in the non-recombining tree model. For a 6-period model, we have 32 old states.
- **new state**: categorized old states.

- **class**: the final states in the recombining tree model. Since they partition the old states, we can view a class as a subset of the old states who end up with the same final state in the recombining tree model.

In our example, we have 6 periods, so the old state refers to the final nodes 0-31 in the non-recombining tree and the recombining tree has classes 0-5. All old states have unique corresponding classes when we compare the 2 tree models, depending on the number of up and down movements they undergo. For instance, class 0 has old state 0 as the single element and class 1 consists of old states 1, 2, 4, 8, 16. In particular, old states 1 and 2 belong to class 1 because 'uuuud' and 'uuudu' both experience 4 upward movements and 1 downward movement. A complete list is:

- class 0 = 0
- class 1 = 1,2,4,8,16
- class 2 = 3,5,6,9,10,12,17,18,20,24
- class 3 = 7,11,13,14,19,21,22,25,26,28
- class 4 = 15,23,27,29,30
- class 5 = 31

To do the transformation, we construct a tree, namely new tree, that has the same structure as the non-recombining one in hand. However, the final states, or new states, in the new tree take different values from the original tree. For recombining purpose, we look into each class and take the average of the damages corresponding to its elements, or new states, in the non-recombining model. Then we assign each new state in the new tree the average damage of its class. For example, new states 3,5,6,9,10,12,17,18,20,24 all have the same value - their average damage in the original tree- as they all belong to class 2 in the recombining tree.

3 Damage

3.1 Inputs

- **tree**: **TreeModel** object in tree.py provides the tree structure that we need for information and step-by-step calculation. Equally important are the probabilities of different nodes and number of periods and states.
- **bau**: **BusinessAsUsual** object in bau.py that provides the business-as-usual scenario of emissions. bau gives us base case information of GHG levels across time if no mitigation is to be done. Based on bau we get realized GHG levels and forcing through forcing.py.

3.2 Attributes

The parameters of **Damage** are taken as attributes as well.

- **tree**: **TreeModel** object in tree.py provides the tree structure that we need for information and step-by-step calculation.
- **bau**: **BusinessAsUsual** object in bau.py that provides the business-as-usual scenario of emissions.

4 DLWDamage

Compared to the Damage class, DLWDamage class has many more parameters that we need to take care of, since it is the subclass designed to encompass typical characteristics, like a constant consumption growth rate, and give concrete mathematical outputs.

4.1 Inputs

- **tree**: **TreeModel** object in tree.py provides the tree structure that we need for information and step-by-step calculation.
- **bau**: **BusinessAsUsual** object in bau.py that provides the business-as-usual scenario of emissions. bau gives us base case information of GHG levels across time if no mitigation is to be done. Based on bau we get realized GHG levels and forcing through forcing.py.
- **cons_growth**: a float that represents the constant consumption growth rate we need to model consumption. The paper states that consumption grows at a rate of about 2% per year. In the example, the model takes the consumption growth rate as an input and uses 0.015.
- **ghg_levels**: an array or list that contains the final GHGs levels for each mitigation scenario. For example, let ghg_levels=[450, 650, 1000]. This is because we assume 3 different mitigation scenarios, under which the maximum levels of GHGs in atmosphere should be 450, 650, 1000 ppm respectively. They individually reflect a strict, a modest, and an ineffective mitigation scenario. We need to do simulation for each of these scenarios.
- **subinterval_len**: a float that represents the length of a typical sub-interval. This is the unit of time over which we calculate the forcing-related variables in forcing.py and define the probability of hitting a Tipping Point, see **_tipping_point_update** in damage_simulation.py. In our example, this number is 5.

4.2 Attributes

- **tree**: **TreeModel** object in tree.py provides the tree structure that we need for information and step-by-step calculation.
- **bau**: **BusinessAsUsual** object in bau.py that provides the business-as-usual scenario of emissions.
- **cons_growth**: a float that represents the constant consumption growth rate we need to model consumption. In our example, this number is 0.015 by assumption in the paper.
- **ghg_levels**: an array that contains the GHGs levels for each ending scenario. For example, let ghg_levels=[450, 650, 1000].
- **dnum**: an integer that represents the number of ending scenarios that we would like to simulate for. For example, when we consider 3 different scenarios of maximum GHGs levels, i.e. **ghg_levels**, dnum = 3.
- **d**: an array of simulated damages. It is an array of dimension base scenarios × number of final states × number of periods with entries being corresponding simulation outcomes.
- **d_rcomb**: an array of adjusted simulated damages for a recombining tree.
- **cum_forcings**: a array of cumulative forcing interpolation coefficients that are used to calculate forcing based mitigation. We get these from **forcing.py**.
- **damage_coefs**: an array of coefficients that are used to interpolate damages.
- **emit_pct**: an array that represents the percentage differences in GHG level changes under each mitigation scenario, compared to the business-as-usual case. We can call this **mitigation percentage**. For example, under the modest mitigation scenario, the maximum ending GHG level is 650. Assume we have GHG levels change from ghg_0 to ghg_T with no mitigation implemented. Then the mitigation percentage corresponding to the modest mitigation scenario should be $1 - \frac{650 - ghg_0}{ghg_T - ghg_0}$

5 Python

First import necessary packages.

```
from __future__ import division, print_function
import numpy as np
from abc import ABCMeta, abstractmethod
from damage_simulation import DamageSimulation
from forcing import Forcing
```

5.1 Damage

The Damage class is a superclass that offers the necessary variables and functions for its subclasses, like DLWDamage, to use.

```
class Damage(object):
        """Abstract damage class for the EZ-Climate model.

        Parameters
        ----------
        tree : `TreeModel` object
                provides the tree structure used
        bau : `BusinessAsUsual` object
                business-as-usual scenario of emissions

        Attributes
        ----------
        tree : `TreeModel` object
                provides the tree structure used
        bau : `BusinessAsUsual` object
                business-as-usual scenario of emissions

        """
```

5.1.1 Methods

```
        __metaclass__ = ABCMeta
        def __init__(self, tree, bau):
                self.tree = tree
                self.bau = bau
```

average_mitigation: an abstract method that returns a 1-D array of the average mitigation for every node in the period.

```
        @abstractmethod
        def average_mitigation(self):
                """The average_mitigation function should return a 1D array of the
                average mitigation for every node in the period.
                """
                pass
```

damage_function: an abstract method that returns a 1-D array of the damages for every node in the period.

```
        @abstractmethod
        def damage_function(self):
```

```
            """The damage_function should return a 1D array of the damages for
            every node in the period.
            """
            pass
```

5.2 DLWDamage

The DLWDamage class is a subclass of Damage initiated with parameters including the TreeModel, the base case information given by the business-as-usual scenario, the growth rate of consumption, the ending GHG levels for different mitigation scenarios, and the arbitrary length of a typical subinterval.

```
class DLWDamage(Damage):
        """Damage class for the EZ-Climate model. Provides the damages from emissions

        Parameters
        ----------
        tree : `TreeModel` object
                provides the tree structure used
        bau : `BusinessAsUsual` object
                business-as-usual scenario of emissions
        cons_growth : float
                constant consumption growth rate
        ghg_levels : ndarray or list
                end GHG levels for each end scenario
        subinterval_len: float
                 represents the length of a typical sub-interval.

        Attributes
        ----------
        tree : `TreeModel` object
                provides the tree structure used
        bau : `BusinessAsUsual` object
                business-as-usual scenario of emissions
        cons_growth : float
                constant consumption growth rate
        ghg_levels : ndarray or list
                end GHG levels for each end scenario
        dnum : int
                number of simulated damage paths
        d : ndarray
                simulated damages
        d_rcomb : ndarray
                adjusted simulated damages for recombining tree
```

```
        cum_forcing : ndarray
                cumulative forcing interpolation coefficients, used to calculate forci
        forcing : `Forcing` object
                class for calculating cumulative forcing and GHG levels
        damage_coefs : ndarray
                interpolation coefficients used to calculate damages

        emit_pct: float? shouldn't it be an array?
                = 1.0 - (self.ghg_levels-self.bau.ghg_start) / bau_emission
                the percentage of the cumulative emission up to the beginning of the p

        cum_forcings :???

        """
        def __init__(self, tree, bau, cons_growth, ghg_levels, subinterval_len):
                super(DLWDamage, self).__init__(tree, bau)
                self.ghg_levels = ghg_levels
                if isinstance(self.ghg_levels, list):
                        self.ghg_levels = np.array(self.ghg_levels)
                self.cons_growth = cons_growth
                self.dnum = len(ghg_levels)
                self.subinterval_len = subinterval_len
                self.cum_forcings = None
                self.d = None
                self.d_rcomb = None
                self.emit_pct = None
                self.damage_coefs = None
```

5.2.1 Methods

_recombine_nodes: transforms information under the **TreeModel** into a recombined model.

- set up the storage space.

```
        def _recombine_nodes(self):
                """Creating damage coefficients for recombining tree. The state re
                separate from a down-up move because in general the two paths wil
                mitigation and therefore of GHG level. A 'recombining' tree is on
                one state to the next through time is nonetheless such that an up
                leads to the same fragility.
                """
                nperiods = self.tree.num_periods
                sum_class = np.zeros(nperiods, dtype=int)
                new_state = np.zeros([nperiods, self.tree.num_final_states], dtype=
```

```
temp_prob = self.tree.final_states_prob.copy()
self.d_rcomb = self.d.copy()
```

- start with the **TreeModel**, determine the old states that belong to the same new state in a recombining tree in the final period. Here are the steps:

  new_state is a matrix. Every row corresponds to a final state in the recombining tree, and its entries are the final states in the non-recombining tree that belong to this new state. For example, for entries in the third row indexed 2, they are all old states in the non-recombining tree model that belong to the same new state 2 in the recombining tree model.
  sum_class is an array whose entries represent the numbers of old states that belong to each new state. For example, in a binomial model with 6 periods, the sum_class is [1 5 10 10 5 1]. This means that the first final state in a recombining tree corresponds to one final state in the non-recombining tree, and the third final state in the recombining tree stems from 10 final states in a recombining tree.

```
for old_state in range(self.tree.num_final_states): #look at final
        temp = old_state
        n = nperiods-2 #last period before recombining
        d_class = 0
        while n >= 0:
                if temp >= 2**n: #modify the lower half of all fin
                        temp -= 2**n
                        d_class += 1
                n -= 1
        sum_class[d_class] += 1
        new_state[d_class, sum_class[d_class]-1] = old_state # slic
'''
the new_state for our model should be something like
 0
 1  2  4  8  16
 3  5  6  9  10 12 17 18 20 24
 7 11  13  14 19 21 22 25 26 28
 15 23 27 29 30
 31
the sum_class should be [1 5 10 10 5 1]
'''
```

- **prob_sum** is an array whose entries represent the sums of probabilities of final states under the non-recombining tree that contribute to the final states under teh recombining tree.

```
sum_nodes = np.append(0, sum_class.cumsum())
prob_sum = np.array([self.tree.final_states_prob[sum_nodes[i]:sum_n
'''
sum_nodes: [ 0, 1, 6, 16, 26, 31, 32]
```

```
                prob_sum: sums up the probabilities of final states
                '''
```

- for each period and each simulated path, we calculate set by set the damage and update the individual states covered with the probability of state 0 in the final period.

  d_sum is an array that stores the probability-weighted sum of the simulated damage for each category.

```
                for period in range(nperiods): #look into each period
                        for k in range(self.dnum): #look into each simulated path
                                d_sum = np.zeros(nperiods) #to store sums of simula
                                old_state = 0 #look at states by jumps/stages
                                for d_class in range(nperiods): #look into each sta
                                        d_sum[d_class] = (self.tree.final_states_pr
                                                                * self.d_
                                        # it is Prob. * damage, damage is got thro
                                        old_state += sum_class[d_class] # moving th
                                        self.tree.final_states_prob[new_state[d_cla
                    #update final_states_prob with the summed prob.

                                for d_class in range(nperiods):
                                        self.d_rcomb[k, new_state[d_class, 0:sum_cl
                # find the probability-weighted average damage

            self.tree.node_prob[-len(self.tree.final_states_prob):] = self.tree.final_s
                    for p in range(1,nperiods-1): #look into intermediate periods
                            nodes = self.tree.get_nodes_in_period(p) #the first and las
                            for node in range(nodes[0], nodes[1]+1): #look into all the
                                    worst_end_state, best_end_state = self.tree.reachab
                                    self.tree.node_prob[node] = self.tree.final_states_
                #update the nodes's prob using the new final_states_prob (always end u
```

_damage_interpolation: determines the interpolation coefficients for **damage_function**.
Here are the steps:

- import stored damage simulation outcomes if not available.

```
        def _damage_interpolation(self):
                """Create the interpolation coefficients used in `damage_function`
                if self.d is None:
                        print("Importing stored damage simulation")
                        self.import_damages()
```

- recombine the nodes, determine **emit_pct** if not available.

```
                self._recombine_nodes()
        #init emit_pct
                if self.emit_pct is None:
```

```
                bau_emission = self.bau.ghg_end - self.bau.ghg_start
                self.emit_pct = 1.0 - (self.ghg_levels-self.bau.ghg_start)
```

- Define the storage space for damage coefficients, and coefficients of the matrix equation for the quadratic relationship in concern.

```
                self.damage_coefs = np.zeros((self.tree.num_final_states, self.tree
                amat = np.ones((self.tree.num_periods, self.dnum, self.dnum))
                bmat = np.ones((self.tree.num_periods, self.dnum)) #period vs simul
```

- Determine the linear interpolation of damages between the 650 and 1000 ppm scenarios.

```
                self.damage_coefs[:, :, -1,  -1] = self.d_rcomb[-1, :, :] # d_romb'
                self.damage_coefs[:, :, -1,  -2] = (self.d_rcomb[-2, :, :] - self.d
        #difference in simulation outcomes/cumulative emission
```

- Solve the quadratic interpolation of damages bwteen 450 and 650 ppm.

```
                amat[:, 0, 0] = 2.0 * self.emit_pct[-2]
                amat[:, 1:, 0] = self.emit_pct[:-1]**2
                amat[:, 1:, 1] = self.emit_pct[:-1]
                amat[:, 0, -1] = 0.0

                for state in range(0, self.tree.num_final_states):
                        bmat[:, 0] = self.damage_coefs[state, :, -1,  -2] * self.em
                        bmat[:, 1:] = self.d_rcomb[:-1, state, :].T #?
                        self.damage_coefs[state, :, 0] = np.linalg.solve(amat, bmat
```

import_damages: import saved simulated damages. File must be saved in 'data' directory under current working directory. Here are the steps:

- save imported values in **d** as the damages for each base scenario and period.
- execute **_damage_interpolation** to interpolate damages in other periods.

```
    def import_damages(self, file_name="simulated_damages"):
            """Import saved simulated damages. File must be saved in 'data' direct
            inside current working directory. Save imported values in `d`.

            Parameters
            ----------
            file_name : str, optional
                    name of file of saved simulated damages

            Raises
            ------
            IOError
                    If file does not exist.

            """
```

```
            from tools import import_csv # import damage from a csv file
            try:
                    d = import_csv(file_name, ignore="#", header=False)
            except IOError as e:
                    import sys
                    print("Could not import simulated damages:\n\t{}".format(e))
                    sys.exit(0)

            n = self.tree.num_final_states
            self.d = np.array([d[n*i:n*(i+1)] for i in range(0, self.dnum)])
    #d is an array with entries being the simulated outcomes for damages in the fi
            self._damage_interpolation()
```

damage_simulation: initiate **DamageSimulation** in damage_simulation.py to simulate damages. Here are the steps:

- use **DamageSimulation** to get simulated damage outcomes for the last period under the defined consumption growth rate. We set the parameters as follows: disaster_tail and peak_temp have different values from the paper

  peak_temp=peakT=9.0.

  disaster_tail=β=12.0, curvature of the tipping point.

  tip_on=True, so we considers the tipping point case.

  temp_map=1, so we adopt the Wagner-Weitzman normal model.

  temp_dist_params=None

  maxh=100.0, so we assume it takes 100 years for temperature to get half way to its maximum value for a given level of GHGs under Pindyck.

  save_simulation=True, so we save the simulation outcomes.

- execute **_damage_interpolation** to determine the coefficients used in **damage_function_node** .

```
    def damage_simulation(self, draws, peak_temp=9.0, disaster_tail=12.0, tip_on=Tru
            temp_map=1, temp_dist_params=None, maxh=100.0, save_simulation=True):
            """Initializion and simulation of damages, given by :mod:`ez_climate.D

            Parameters
            ----------
            draws : int
                    number of Monte Carlo draws
            peak_temp : float, optional
                    tipping point parameter
            disaster_tail : float, optional
                    curvature of tipping point
            tip_on : bool, optional
```

```
            flag that turns tipping points on or off
        temp_map : int, optional
            mapping from GHG to temperature
            * 0: implies Pindyck displace gamma
            * 1: implies Wagner-Weitzman normal
            * 2: implies Roe-Baker
            * 3: implies user-defined normal
            * 4: implies user-defined gamma
        temp_dist_params : ndarray or list, optional
            if temp_map is either 3 or 4, user needs to define the distrib
        maxh : float, optional
            time paramter from Pindyck which indicates the time it takes f
            way to its max value for a given level of ghg
        cons_growth : float, optional
            yearly growth in consumption
        save_simulation : bool, optional
            True if simulated values should be save, False otherwise

        Returns
        -------
        ndarray
            simulated damages

        """
        # get simulated damage from damge-simulation.py
        ds = DamageSimulation(tree=self.tree, ghg_levels=self.ghg_levels, peak_t
                              disaster_tail=disaster_tail, tip_on=tip_on, temp
                              temp_dist_params=temp_dist_params, maxh=maxh, co
        print("Starting damage simulation..")
        self.d = ds.simulate(draws, write_to_file = save_simulation)
        print("Done!")
        self._damage_interpolation()
        return self.d
```

_forcing_init: use forcing.py to give the cumulative forcing up to each node under each simulation scenarios.

- determine mitigation percentages **emit_pct** via bau.py if not available.

```
def _forcing_init(self):
    """Initialize `Forcing` object and cum_forcings used in calculatin
    if self.emit_pct is None:
            bau_emission = self.bau.ghg_end - self.bau.ghg_start
            self.emit_pct = 1.0 - (self.ghg_levels-self.bau.ghg_start)
```

- for each simulation path, figure out the nodes in each period and calculate cumulative forcing up to the nodes with **forcing_at_node** in forcing.py.

```
            self.cum_forcings = np.zeros((self.tree.num_periods, self.dnum)) #
            mitigation = np.ones((self.dnum, self.tree.num_decision_nodes)) * s
        #mitigaion is indexed with number of path (row) and number of nodes(colum

            for i in range(0, self.dnum): #look into each simulation
                    for n in range(1, self.tree.num_periods+1):
                            node = self.tree.get_node(n, 0) #the node of state 0 in
                            self.cum_forcings[n-1, i] = Forcing.forcing_at_node(miti
```

_forcing_based_mitigation: based on the cumulative forcings under the baseline scenarios, interpolate the mitigation level for the given cumulative forcing. Here are the steps:

- Figure out where the level of forcings given falls in between the baseline scenarios.
- Calculate the corresponding mitigation level based on those of the baseline scenarios.

```
    def _forcing_based_mitigation(self, forcing, period):
            """Calculation of mitigation based on forcing up to period. Interpolat
            with the constant degree of mitigation consistent with the damage simu
            """
            # this whole function is based on a new theory
            p = period - 1
            if forcing > self.cum_forcings[p][1]:
                    weight_on_sim2 = (self.cum_forcings[p][2] - forcing) / (self.cum
                    weight_on_sim3 = 0
            elif forcing > self.cum_forcings[p][0]:
                    weight_on_sim2 = (forcing - self.cum_forcings[p][0]) / (self.cum
                    weight_on_sim3 = (self.cum_forcings[p][1] - forcing) / (self.cum
            else:
                    weight_on_sim2 = 0
                    weight_on_sim3 = 1.0 + (self.cum_forcings[p][0] - forcing) / sel

            return weight_on_sim2 * self.emit_pct[1] + weight_on_sim3*self.emit_pct[
```

average_mitigation_node: calculate the average fractional-mitigation level up to a given node. Here are the steps:

- Find the path from the origin to the given node, compute the mitigation during each period along this path as well as the total emission under the non-mitigation scenario.
- Calculate the average mitigation up to this given node: $\bar{m} = \frac{\sum_{i \in \text{period}} m_i \times \text{length of period } i}{\text{total emission}}$, with the help of **TreeModel** and **BusinessAsUsual**.

```
    def average_mitigation_node(self, m, node, period=None):
            """Calculate the average mitigation up to a given node.

            Parameters
            ----------
```

```
        m : ndarray or list
                array of mitigation
        node : int
                node for which average mitigation is to be calculated for
        period : int, optional
                the period the node is in

        Returns
        -------
        float
                average mitigation

        """
        if period == 0:
                return 0
        if period is None:
                period = self.tree.get_period(node)
        state = self.tree.get_state(node, period)
        path = self.tree.get_path(node, period)
        new_m = m[path[:-1]] # mitigation on the path until this node

        period_len = self.tree.decision_times[1:period+1] - self.tree.decision_t
        bau_emissions = self.bau.emission_by_decisions[:period] #emission level
        total_emission = np.dot(bau_emissions, period_len) #total emission: sum
        ave_mitigation = np.dot(new_m, bau_emissions*period_len) # mitigation f
        return ave_mitigation / total_emission # average mitigation until node
```

average_mitigation: calculate the average mitigation for all nodes up to a given period, via **get_num_nodes_period** and **get_node** of the **TreeModel** and **average_mitigation_node**.

```
    def average_mitigation(self, m, period):
        """Calculate the average mitigation for all nodes in a period.

        m : ndarray or list
                array of mitigation
        period : int
                period to calculate average mitigation for

        Returns
        -------
        ndarray
                average mitigations

        """
        nodes = self.tree.get_num_nodes_period(period) #number of nodes for a g
        ave_mitigation = np.zeros(nodes)
```

```
        for i in range(nodes):
                node = self.tree.get_node(period, i)
                ave_mitigation[i] = self.average_mitigation_node(m, node, period
        return ave_mitigation
```

_ghg_level_node: based on the defined tree structure, business-as-usual model, and sub-interval length, calculate the GHG level at a given node under given mitigation action, via **Forcing.ghg_level_at_node** in forcing.py.

```
    def _ghg_level_node(self, m, node):
        return Forcing.ghg_level_at_node(m, node, self.tree, self.bau, self.subi
```

_ghg_level_period: calculates the GHG levels corresponding to given mitigation for given nodes or nodes of a given period.

Find the nodes with the target period with **get_nodes_in_period**. Then find the GHG levels for the all the nodes with **_ghg_level_node**.

```
    def ghg_level_period(self, m, period=None, nodes=None):
        """Calculate the GHG levels corresponding to the given mitigation.
        Need to provide either `period` or `nodes`.

        Parameters
        ----------
        m : ndarray or list
                array of mitigation
        period : int, optional
                what period to calculate GHG levels for
        nodes : ndarray or list, optional
                the nodes to calculate GHG levels for

        Returns
        -------
        ndarray
                GHG levels

        """
        if nodes is None and period is not None:
                start_node, end_node = self.tree.get_nodes_in_period(period)
                if period >= self.tree.num_periods:
                        add = end_node-start_node+1
                        start_node += add
                        end_node += add
                nodes = np.array(range(start_node, end_node+1))
        if period is None and nodes is None:
                raise ValueError("Need to give function either nodes or the peri
```

```
        ghg_level = np.zeros(len(nodes))
        for i in range(len(nodes)):
                ghg_level[i] = self._ghg_level_node(m, nodes[i])
        return ghg_level
```

ghg_level: calculates the GHGs levels for all the states up to the given period. The code has the following steps:

- create storage space according to the given period.
- for each period, find the corresponding nodes and find their GHGs levels with **_ghg_level_period**.

```
    def ghg_level(self, m, periods=None):
        """Calculate the GHG levels for more than one period.

        Parameters
        ----------
        m : ndarray or list
                array of mitigation
        periods : int, optional
                number of periods to calculate GHG levels for

        Returns
        -------
        ndarray
                GHG levels

        """
        #create stroage space for ghg_level for every possible states
        if periods is None:
                periods = self.tree.num_periods-1
        if periods >= self.tree.num_periods:
                ghg_level = np.zeros(self.tree.num_decision_nodes+self.tree.num_
        else:
                ghg_level = np.zeros(self.tree.num_decision_nodes)
        # for each period, find the right start and end node within this perio
        for period in range(periods+1):
                start_node, end_node = self.tree.get_nodes_in_period(period)
                if period >= self.tree.num_periods:
                        add = end_node-start_node+1
                        start_node += add
                        end_node += add
                nodes = np.array(range(start_node, end_node+1))
                ghg_level[nodes] = self.ghg_level_period(m, nodes=nodes)
        return ghg_level
```

_damage_function_node: Calculate the damage at a given node, based on the given miti-

gation level. Here are the steps:

- Implement **_damage_interpolation** and **_forcing_init** to get the cumulative forcings under the baseline scenarios and the damage interpolation functions if necessary.

```
def _damage_function_node(self, m, node):
        """Calculate the damage at any given node, based on mitigation act
        if self.damage_coefs is None:
                self._damage_interpolation()
        if self.cum_forcings is None:
                self._forcing_init()
        if node == 0:
                return 0.0
```

- Determine the period, cumulative forcing, GHG level and force_based mitigation for the given node under specified mitigation level.

```
period = self.tree.get_period(node)
forcing, ghg_level = Forcing.forcing_and_ghg_at_node(m, node, self.
force_mitigation = self._forcing_based_mitigation(forcing, period)
```

- Define the penalty **ghg_extension** if the concentration goes below pre-industrial levels.

```
ghg_extension = 1.0 / (1 + np.exp(0.05*(ghg_level-200)))
```

- Determine the reachable final states as well as their probabilities for the given node.

```
worst_end_state, best_end_state = self.tree.reachable_end_states(no
probs = self.tree.final_states_prob[worst_end_state:best_end_state+
```

- If the forcing_based mitigation level is lower than that of the 650 ppm case, use the linear interpolation function to determine the damage level.

```
if force_mitigation < self.emit_pct[1]:
        damage = (probs *(self.damage_coefs[worst_end_state:best_en
                        + self.damage_coefs[worst_end_state:best_e
```

- If the forcing_based mitigation level is between those of the 450 and 650 ppm cases, use the quadratic interpolation function to determine the damage level.

```
elif force_mitigation < self.emit_pct[0]:
        damage = (probs * (self.damage_coefs[worst_end_state:best_e
                        + self.damage_coefs[worst_end_state:best_
                        + self.damage_coefs[worst_end_state:best_
```

- If the forcing_based mitigation level is higher than that of the 450 ppm, use Equation 7 and introduce the penalty instead.

```
else:
        damage = 0.0
        i = 0
        for state in range(worst_end_state, best_end_state+1):
```

```
                if self.d_rcomb[0, state, period-1] > 1e-5:
                        deriv = 2.0 * self.damage_coefs[state, peri
                                        + self.damage_coefs[state,
                        decay_scale = deriv / (self.d_rcomb[0, stat
                        dist = force_mitigation - self.emit_pct[0]
                                / (np.log(0.5) * decay_scale)
                        damage += probs[i] * (0.5**(decay_scale*dis
                i += 1

        return (damage / probs.sum()) + ghg_extension
```

damage_function: calculate the damage for every node in a given period, based on mitigation action input **m**. Find the nodes in the given period, use **_damage_function_node** to determine resulted damage at each node one by each time.

```
def damage_function(self, m, period):
        """Calculate the damage for every node in a period, based on mitigatio

        Parameters
        ----------
        m : ndarray or list
                array of mitigation
        period : int
                period to calculate damages for

        Returns
        -------
        ndarray
                damages

        """
        nodes = self.tree.get_num_nodes_period(period)
        damages = np.zeros(nodes)
        for i in range(nodes):
                node = self.tree.get_node(period, i)
                damages[i] = self._damage_function_node(m, node)
        return damages
```