Documentation for Utility.py

Yili Yang

June 2017

1 Introduction

Our model uses preference specification suggested by Epstein and Zin that allows different rates of substitution across time and states. In an Epstein-Zin utility framework, the agent maximizes at each time t:

$$U_t = [(1-\beta)c_t^{\rho} + \beta[\mu_t(\tilde{U}_{t+1})^{\rho}]]^{\frac{1}{\rho}} \tag{1}$$

where $\mu_t(\tilde{U}_{t+1})$ is the certainty-equivalent of future lifetime utility, based on the agent's information at time t, which is given by:

$$\mu_t(\tilde{U}_{t+1}) = (E_t[U_{t+1}^{\alpha}])^{\frac{1}{\alpha}} \tag{2}$$

In this specification,$(1-\beta)/\beta$ is the pure rate of **time preference**. The parameter ρ measures the agent's willingness to substitute consumption across time. The higher is ρ, the more willing the agent is to substitute consumption across time.
The elasticity of intertemporal substitution is given by $\sigma = 1/(1-\rho)$
The α captures the agent's willingness to substitute consumption across (uncertain) future consumption streams. The higher is α, the more willing the agent is to substitute consumption across states of nature at a given point in time.
The coefficient of relative risk aversion at a given point in time is $\gamma = (1-\alpha)$. This added flexibility allows calibration across states of nature and time.
put (2) in to (1) we get:

$$U_0 = [(1-\beta)c_0^{\rho} + \beta(E_t[U_1^{\alpha}])^{\frac{\rho}{\alpha}}]^{\frac{1}{\rho}} \tag{3}$$

$$U_t = [(1-\beta)c_t^{\rho} + \beta(E_t[U_{t+1}^{\alpha}])^{\frac{\rho}{\alpha}}]]^{\frac{1}{\rho}}, \text{for } t \in \{1, 2, \ldots, T-1\} \tag{4}$$

In the final period, which in our base case is the period starting in 2400, the agent will receive the utility from all consumption from time T forward. Given our assumption that all uncertainty has already been resolved at this point, consumption grows at a constant rate g from T through infinity (i.e. $c_t = c_T(1+g)^{t-T}$ for $t \geq T$) and produces a utility of:

$$U_T = [\frac{1-\beta}{1-\beta(1+g)^{\rho}}]^{1/\rho} c_T \tag{5}$$

The utility of the final stage can also be separated into two parts: the part that is generated by consumption on T and the part is generated in the future, i.e. the certainty-equivalent term. While the certainty-equivalent term in the period before is defined by equation (2). The certainty-equivalent term in the final stage is

$$U_T - (1-\beta)c_T^\rho \tag{6}$$

with

$$c_0 = \bar{c_0}(1-\kappa_0(x_0)) \tag{7}$$

$$c_t = \bar{c_t}(1 - D_t(X_t, \theta_t) - \kappa_t(x_t)) \text{ for } t \in \{1, 2, \ldots, T-1\} \tag{8}$$

$$c_T = \bar{c_T}(1 - D_T(X_T, \theta_T)) \tag{9}$$

where D is the climate damage function, κ is the cost function and X_t is the cumulative mitigation level depends on the level of mitigation in each period from 0 to t, which is given by:

$$X_t = \frac{\sum_{s=0}^{t} g_s x_s}{\sum_{s=0}^{t} g_s} \tag{10}$$

2 Marginal Utility

For sensitivity analysis, we'd like to find the change of utility given some change in consumption i.e. the marginal utility. There are two kinds of consumption which will contribute to the utility:

- the consumption at this time t: c_t
- the consumption at the next time: $(E_t[U_{t+1}^\alpha])^{1/\alpha}$

Now we discuss their contribution separately:
For the first kind, the marginal utility w.r.t. c_t, we consider our utility is a function with one variable c_t as $U_t(c_t)$. For $t \leq T-1$, the formulation is (4), to find margin w.r.t c_t, We take derivative:

$$dU_t = \rho(1-\beta)c_t^{\rho-1}\frac{1}{\rho}[(1-\beta)c_t^\rho + \beta(E_t[U_{t+1}^\alpha])^{\frac{\rho}{\alpha}}]^{\frac{1}{\rho}-1}dc_t \tag{11}$$

$$\Rightarrow dU_t = (1-\beta)c_t^{\rho-1}[(1-\beta)c_t^\rho + \beta(E_t[U_{t+1}^\alpha])^{\frac{\rho}{\alpha}}]^{\frac{1}{\rho}-1}dc_t \tag{12}$$

And for the final period,

$$\rho U_T^{\rho-1}dU_T = \frac{(1-\beta)}{1-\beta(1+g)^\rho}\rho c_T^{\rho-1}dc_T \tag{13}$$

$$\Rightarrow dU_T = \frac{(1-\beta)(\frac{u_T}{c_T})^{1-\rho}}{1-\beta(1+g)^\rho}dc_T \tag{14}$$

We are not sure why he took the derivative in this way, rather than taking the coefficient directly. Then the second kind, the marginal utility w.r.t. certainty-equivalent of future lifetime utility:

For an arbitrary $U_t, t \in \{0, 1, 2, \ldots, T-2\}$ the next stage is unknown but it's either up or down, so we denote the 'up' utility as U_{t_1} with probability p_1, constant consumption c_{t_1} and 'down' as U_{t_2} with probability $p_2 = 1 - p_1$, constant consumption c_{t_2}. According to equation (4), they can be written as:

$$U_{t_i} = [(1-\beta)c_{t_i}^{\rho} + ce_{t_i}]^{1/\rho} \text{ where } \ i = 1, 2 \tag{15}$$

where the $ce_{t_1} = \beta(E_{t_1}[U_{t_1+1}^{\alpha}])^{\frac{\rho}{\alpha}}$ here stands for the adjusted certainty-equivalence term in the 'up' or 'down' stage.
For this marginal utility in the last period before end, denoted as $T-1$, when we get full information, the future utility is known to us and given by

$$E_{T-1}[U_T] = U_T \tag{16}$$

Then the marginal utility of U_{T-1} w.r.t. the consumption in the next period is given by:

$$U_{T-1}^{\rho} = (1-\beta)c_{T-1}^{\rho} + \beta U_T^{\rho} \tag{17}$$

$$U_{T-1}^{\rho} = (1-\beta)c_{T-1}^{\rho} + \frac{\beta(1-\beta)}{1-\beta(1+g)^{\rho}}c_T^{\rho} \tag{18}$$

$$\Rightarrow \rho U_{T-1}^{\rho-1}\frac{dU_{T-1}}{dc_T} = \rho\frac{\beta(1-\beta)}{1-\beta(1+g)^{\rho}}c_T^{\rho-1} \tag{19}$$

$$\Rightarrow \frac{dU_{T-1}}{dc_T} = \frac{\beta(1-\beta)}{1-\beta(1+g)^{\rho}}c_T^{\rho-1}U_{T-1}^{1-\rho} \tag{20}$$

Finally, for the final stage, we can consider it as the same with $p_1 = 1$ and new consumption.
Now we can derive the formula for the second kind:
We consider the utility is a function with only one variable c_{t_i} which means we only consider the effects made by changing one states' (up or down) consumption constant. Using 'up' as an example:

$$ce = \beta[\mu_t(\tilde{U}_{t+1})]^{\rho} \tag{21}$$

put (15) in to (4):

$$U_t = [(1-\beta)c_t^{\rho} + \beta(U_{t_1}^{\alpha}p_1 + U_{t_2}^{\alpha}p_2)^{\frac{\rho}{\alpha}}]^{1/\rho} \tag{22}$$

to perform derivation on (22) w.r.t. the next consumption, we need to perform derivation on both U_{t_1} and U_{t_2}. Let $f(U_{t_1})$ denote $U_{t_1}^{\alpha}p_1$ then

$$U_t = [(1-\beta)c_t^{\rho} + \beta(f(U_{t_1}) + f(U_{t_2}))^{\frac{\rho}{\alpha}}]^{1/\rho} \tag{23}$$

and

$$\frac{df(U_{t_1})}{dc_{t_1}} = \frac{dU_{t_1}}{dc_{t_1}}\frac{df(U_{t_1})}{dU_{t_1}} \tag{24}$$

$$\frac{dU_{t_1}}{dc_{t_1}} = \frac{1}{\rho}[(1-\beta)c_{t_1}^{\rho} + ce_{t_1}]^{1/\rho-1}\rho(1-\beta)c_{t_1}^{\rho-1} \tag{25}$$

$$\frac{df(U_{t_1})}{dU_{t_1}} = \alpha[(1-\beta)c_{t_1}^{\rho} + ce_{t_1}]^{(\alpha-1)/\rho}p_1 \tag{26}$$

$$\Rightarrow \frac{df(U_{t_1})}{dc_{t_1}} = (1-\beta)c_{t_1}^{\rho-1}\alpha p_1[(1-\beta)c_{t_1}^{\rho} + ce_{t_1}]^{\alpha/\rho-1} \tag{27}$$

then

$$\frac{dU_t}{dc_{t_1}} = \frac{df(U_{t_1})}{dc_{t_1}} \frac{dU_t}{df(U_{t_1})} \tag{28}$$

$$\frac{dU_t}{df(U_{t_1})} = (\sum f(U_{t_i}))^{\frac{\rho}{\alpha}-1} \frac{\rho}{\alpha} \beta (1/\rho) U_t^{\frac{1}{\rho}-1} \tag{29}$$

$$= (\sum f(U_{t_i}))^{\frac{\rho}{\alpha}-1} \frac{\beta}{\alpha} U_t^{\frac{1}{\rho}-1} \tag{30}$$

Then the marginal utility for each node in a period is the sum of all the possible margins:

- the marginal utility w.r.t the consumption today.
- the marginal utitliy w.r.t the consumption on the next 'up' stage (if exists)
- the marginal utility w.r.t the consumption on the next 'down' stage (if exists)
- the marginal utlilty w.r.t the consumption on the one and only next stage (if exists)

for **information times**, the utility is like (23), where margin comes from change of c_t, c_{t_1}, c_{t_2}, and the margin is the sum of first 3 items above.
For the $T-1$ stage, the next consumption c_T is known and the margin is from c_{T-1}, c_T, which is the first and forth items above.
for the **final period** and the periods between decision times, there is only one state remain, and the margin only comes from the change of the first and forth items above.

3 EZUtility Class

This is a class to calculate the EZ-utility.

3.1 Inputs and Outputs

Inputs:

- **tree** : ('TreeModel' object) tree structure used.
- **damage** : ('Damage' object) class that provides damage methods.
- **cost** : ('Cost' object) class that provides cost methods.
- **period_len** : (a float) that represents subinterval length, the value in the example is 5.
- **eis** : $\sigma = 1/(1-\rho)$, (an optional float) that represents the elasticity of intertemporal substitution.
- **ra** : $\gamma = 1-\alpha$, (an optional float) that represents the risk-aversion.

- **time_pref** : $(1-\beta)/\beta$, (an optional float) that represents the pure rate of time preference.
- **add_penalty_cost** : (boolean) not been used in the code.
- **max_penalty** : (float) not been used in the code
- **penalty_scale** : (float) not been used in the code

Outputs: The main function of this class is **_utility_generator** which can generate the EZ-utility of each node as well as consumption, cost and certainty-equivalent.

3.2 Attributes

- **tree** : ('TreeModel' object) tree structure used
- **damage** : ('Damage' object) class that provides damage methods
- **cost** : ('Cost' object) class that provides cost methods
- **period_len** : (float) subinterval length
- **decision_times** : (ndarray) attr from tree object, array of years, in the future, when decisions will be made
- **cons_growth** : (float) attr from damage object, consumption growth. In our example, this value is 0.015.
- **growth_term** : (float) 1 + cons_growth
- **r** : (float) the parameter ρ in the introduction
- **a**: (float) the parameter α in the introduction
- **b**: (float) the parameter β in the introduction

3.3 Methods

_end_period_utility: private function to calculate the utility on the final stage. (period T) Using the equation (5) to get the utility on each node within the final period, where c_T is from equation (9) given damage from simulation.
Inputs:

- **m**: (ndarray) mitigation level on each node
- **utility_tree** : (BigStorageTree Object) place to store the calculated utility
- **cons_tree** : (BigStorageTree Object) place to store the calculated consumption
- **cost_tree** : (SmallStorageTree Object) place to store the calculated cost

Outputs: returns None but will modify the three trees in the input.

```
def _end_period_utility(self, m, utility_tree, cons_tree, cost_tree):
        """Calculate the terminal utility."""
        period_ave_mitigation = self.damage.average_mitigation(m, self.tree.num_
        period_damage = self.damage.damage_function(m, self.tree.num_periods)
        damage_nodes = self.tree.get_nodes_in_period(self.tree.num_periods) # av

        period_mitigation = m[damage_nodes[0]:damage_nodes[1]+1] #storage space
        period_cost = self.cost.cost(self.tree.num_periods, period_mitigation, p

        continuation = (1.0 / (1.0 - self.b*(self.growth_term**self.r)))**(1.0/s

        cost_tree.set_value(cost_tree.last_period, period_cost)
        period_consumption = self.potential_cons[-1] * (1.0 - period_damage)
        period_consumption[period_consumption<=0.0] = 1e-18
        cons_tree.set_value(cons_tree.last_period, period_consumption) # set the
        utility_tree.set_value(utility_tree.last_period, (1.0 - self.b)**(1.0/se
```

_end_period_marginal_utility: private function to calculate the marginal utility w.r.t. the consumption function ((11) and (13))and marginal utility w.r.t the consumption next period on the last period before final (17), $T-1$, when we get the full information.
Inputs:

- **mu_tree_0** : (BigStorageTree Object) place to store the calculated first type margin.
- **mu_tree_1**:(BigStorageTree Object) place to store the calculated second type margin.
- **ce_tree**: (BigStorageTree Object) place to store the calculated adjusted certainty-equivalence.
- **utility_tree**: (BigStorageTree Object) place to store the calculated utility
- **cons_tree**: (BigStorageTree Object) place to store the calculated consumption

Outputs: like the function above, it returns None but modifies the storage tree objects in the Input.

```
def _end_period_marginal_utility(self, mu_tree_0, mu_tree_1, ce_tree, utility_tr
        """Calculate the terminal marginal utility."""
        # the utility of the final state can be seperated into two parts: the f
        # the U_T can be wrote in to U_T = [(1-beta)c_T^rau+(1-beta)/(1-beta*(
        # the margin is future - now which is calculated by following codes.
        ce_term = utility_tree.last**self.r - (1.0 - self.b)*cons_tree.last**sel
        ce_tree.set_value(ce_tree.last_period, ce_term)

        mu_0_last = (1.0 - self.b)*(utility_tree[utility_tree.last_period-self.p
        mu_tree_0.set_value(mu_tree_0.last_period, mu_0_last)
        mu_0 = self._mu_0(cons_tree[cons_tree.last_period-self.period_len], ce_t
        mu_tree_0.set_value(mu_tree_0.last_period-self.period_len, mu_0)
```

```
next_term = self.b * (1.0 - self.b) / (1.0 - self.b * self.growth_term**
mu_1 = utility_tree[utility_tree.last_period-self.period_len]**(1-self.r
mu_tree_1.set_value(mu_tree_1.last_period-self.period_len, mu_1)
```

_certain_equivalence: Caclulate certainty equivalence utility. If we are between decision nodes, i.e. no branching, then certainty equivalent utility at time period depends only on the utility next period given information known today.
Otherwise the certainty equivalent utility is the probability-weighted sum of next period utility over the partition reachable from the state.

$$E_t[U_{t+1}^{\alpha}]^{1/\alpha} = \sum_{state \in \mathbb{K}} [(U_{t+1}^{\alpha}|state)\mathbf{p}(state)]^{1/\alpha} \tag{31}$$

where $\mathbb{K}$ is a set of all the possible states.
Inputs:

- **period**: (int) time now when we calculate the ce term.
- **damage_period**: (int) last time one item split in to two. For example, if the we are dealing with BigStorageTree[5,[15,30,40,50]], then the damage_period for 35 is 30.
- **utility_tree**: (BigStorageTree Object) place to store the calculated Utility

For example, if $bst = BigStorageTree(5, [0, 15, 20, 35, 50])$ then period can be any time that is divisible by 5 and less or equal to 50. Then if the period is 10, then the damage_period is 0. If the period is 30 then the damage_period is 20. Also the utility tree not only calculates the utility for the given period, but also calculates the utilities for periods to follow.

Outputs:

- $\mathbf{cert}_equiv : ('BigStorageTree'Object)thecertainty - equivanlenceoftheinput'period'.$

```
def _certain_equivalence(self, period, damage_period, utility_tree):
        """Caclulate certainty equivalence utility. If we are between decision
        then certainty equivalent utility at time period depends only on the u
        given information known today. Otherwise the certainty equivalent util
        weighted sum of next period utility over the partition reachable from
        """
        if utility_tree.is_information_period(period):
                damage_nodes = self.tree.get_nodes_in_period(damage_period+1)
                probs = self.tree.node_prob[damage_nodes[0]:damage_nodes[1]+1]
                even_probs = probs[::2]
                odd_probs = probs[1::2]
                even_util = ((utility_tree.get_next_period_array(period)[::2])**
                odd_util = ((utility_tree.get_next_period_array(period)[1::2])**
                ave_util = (even_util + odd_util) / (even_probs + odd_probs)
                cert_equiv = ave_util**(1.0/self.a)
        else:
                # no branching implies certainty equivalent utility at time pe
```

```
                # the utility next period given information known today
                cert_equiv = utility_tree.get_next_period_array(period)

        return cert_equiv
```

_mu_0: a private which calculates the marginal utility with respect to consumption function on this period. It is been proved in equation (11). **Inputs**:

- **cons**: (float) consumption today when calculating the marginal utility.
- **ce_term**: (float) modified certainty equivalence term $ce_term = \beta \mathbf{E}(U_{t+1})^{\rho}$

Outputs:

- **mu_0** : (float) the marginal utility with respect to consumption this period.

```
    def _mu_0(self, cons, ce_term):
        """Marginal utility with respect to consumption function."""
        t1 = (1.0 - self.b)*cons**(self.r-1.0)
        t2 = (ce_term - (self.b-1.0)*cons**self.r)**((1.0/self.r)-1.0)
        return t1 * t2
```

_mu_1: a private which calculates the marginal utility with respect to one potential consumption ('up' or 'down')in next period, the output is like equation (28).
Inputs:

- **cons**: (float) consumption today when calculating the marginal utility.
- **prob**: (float) probability p_1 in the introduction, the probability of a state transfers to the 'up' state.
- **cons_1**,**cons_2** : (float) the consumption for the 'up' and 'down' state respectively.
- **ce_1**,**ce_2** :(float) the certainty-equivalence term for the 'up' and 'down' state respectively.
- potential bug **do_print**: (bool) not been used.

Outputs:

- **mu_1**: (float) the marginal utility with respect to the potential consumption on next period.

```
    def _mu_1(self, cons, prob, cons_1, cons_2, ce_1, ce_2, do_print=False):
        """ marginal utility with respect to consumption next period."""
        t1 = (1.0-self.b) * self.b * prob * cons_1**(self.r-1.0)
        t2 = (ce_1 - (self.b-1.0) * cons_1**self.r )**((self.a/self.r)-1)
        t3 = (prob * (ce_1 - (self.b*(cons_1**self.r)) + cons_1**self.r)**(self.
                + (1.0-prob) * (ce_2 - (self.b-1.0) * cons_2**self.r)**(self.a/
        t4 = prob * (ce_1-self.b * (cons_1**self.r) + cons_1**self.r)**(self.a/s
                + (1.0-prob) * (ce_2 - self.b * (cons_2**self.r) + cons_2**self
        t5 = (self.b * t4**(self.r/self.a) - (self.b-1.0) * cons**self.r )**((1.
```

```
        return t1 * t2 * t3 * t5
```

_mu_2: A private function to calculate the marginal utility of the inter-decision-periods (definition in documentation for storage_tree). If we are dealing with period that we are not given information and we are not making decision, there is no uncertainty w.r.t. the consumption next time, then the margin is only generated by the constant increase of the future consumption.

Inputs:

- **cons**: (float) consumption c_t of today when the margin is calculated
- **prev_cons**: (float) consumption of the next period c_{t+1} provided by the consumption tree.
- **ce_term**: (float) ce term of the next period ce_{t+1} provided by ce tree where $ce_t = \beta \mathbf{E}(U_{t+1})^\rho$

$$U_t = [(1-\beta)c_t^\rho + \beta[(1-\beta)c_{t+1}^\rho + \beta ce_{t+1}^\rho]]^{1/\rho} \tag{32}$$

$$\Rightarrow \frac{dU_t}{dc_{t+1}} = (1-\beta)\beta c_{t+1}^{\rho-1}[(1-\beta)c_t^\rho + \beta(1-\beta)c_{t+1}^\rho + \beta^2 ce_{t+1}^\rho]^{1/\rho-1} \tag{33}$$

Outputs:

- **mu_2**: (float) the marginal utility w.r.t. the inter-decision-periods

```
    def _mu_2(self, cons, prev_cons, ce_term):
            """Marginal utility with respect to last period consumption."""
            t1 = (1.0-self.b) * self.b * prev_cons**(self.r-1.0)
            t2 = ((1.0 - self.b) * cons**self.r - (self.b - 1.0) * self.b \
                 * prev_cons**self.r + self.b * ce_term)**((1.0/self.r)-1.0)
            return t1 * t2
```

_utility_generator: main function of the class which generates the utility and calculates the consumption, cost and certain-equivalent on each node as by-products. The focus here is to calculate the right consumption in this period and then put it to equation (4) where certain-equivalence is given by **_certain_equivalence**.
There are 2 situations for calculating consumption:

- For decision time, consumption= *the init consumption*$^{1+constant\_growth\_rate \times years} \times (1 - damage) \times (1 - cost)$
- For time between decision times, we smooth the consumptions through years. For example: if the consumption grows 10 percent during the next 25 years, then the consumption grows $1.1^{5/25} - 1 = 1.1^{0.2} - 1 = 0.019 = 1.9\%$ in the next 5 years. Based on that, we can calculate the utility on every node.

Inputs:

- **mu_tree_0** : (BigStorageTree Object) place to store the calculated first type margin.

- **mu_tree_1**:(BigStorageTree Object) place to store the calculated second type margin.
- **utility_tree**: (BigStorageTree Object) place to store the calculated utility
- **cons_tree**: (BigStorageTree Object) place to store the calculated consumption
- **ce_tree**: (BigStorageTree Object) place to store the calculated ce term

Outputs: like the function above, it returns None but modifies the storage tree objects in the Input. With which, $ce_t = \beta \mathbf{E}(U_{t+1})^{\rho}$

```
    def _utility_generator(self, m, utility_tree, cons_tree, cost_tree, ce_tree, con
            """Generator for calculating utility for each utility period besides t
            # there are two kinds of periods: make dicision/not make dicision.
            periods = utility_tree.periods[::-1]

            for period in periods[1:]:
                    damage_period = utility_tree.between_decision_times(period)
                    cert_equiv = self._certain_equivalence(period, damage_period, ut

                    if utility_tree.is_decision_period(period+self.period_len):
                            damage_nodes = self.tree.get_nodes_in_period(damage_peri
                            period_mitigation =  m[damage_nodes[0]:damage_nodes[1]+1
                            period_ave_mitigation = self.damage.average_mitigation(m
                            period_cost = self.cost.cost(damage_period, period_mitig
                            period_damage = self.damage.damage_function(m, damage_pe
                            cost_tree.set_value(cost_tree.index_below(period+self.pe

                    period_consumption = self.potential_cons[damage_period] * (1.0 -
                    period_consumption[period_consumption <= 0.0] = 1e-18

                    if not utility_tree.is_decision_period(period):
                            #if not a decision time
                            next_consumption = cons_tree.get_next_period_array(perio
                            segment = period - utility_tree.decision_times[damage_pe
                            interval = segment + utility_tree.subinterval_len
                            # if the next period is a decision period
                            if utility_tree.is_decision_period(period+self.period_le
                                    if period < utility_tree.decision_times[-2]:
                                            next_cost = cost_tree[period+self.period
                                            next_consumption *= (1.0 - np.repeat(per
                                            next_consumption[next_consumption<=0.0]
                            # if the information is not gained in next period, the
                            if period < utility_tree.decision_times[-2]:
                                    temp_consumption = next_consumption/np.repeat(pe
                                    period_consumption = np.sign(temp_consumption)*(
                                                    * np.repeat(period_consumpt
```

```
                else:
                        temp_consumption = next_consumption/period_consu
                        period_consumption = np.sign(temp_consumption)*(
                                * period_consumption


        if period == 0:
                period_consumption += cons_adj

        ce_term = self.b * cert_equiv**self.r
        ce_tree.set_value(period, ce_term)
        cons_tree.set_value(period, period_consumption)
        u = ((1.0-self.b)*period_consumption**self.r + ce_term)**(1.0/se
        yield u, period
```

utility: take the mitigation level on each node and run **_utility_generator** to return utility tree and consumption, cost, c_e tree.

```
    def utility(self, m, return_trees=False):
        """Calculating utility for the specific mitigation decisions `m`.

        Parameters
        ----------
        m : ndarray or list
                array of mitigations
        return_trees : bool
                True if methid should return trees calculated in producing the

        Returns
        -------
        ndarray or tuple
                tuple of `BaseStorageTree` if return_trees else ndarray with u

        Examples:
        ---------
        Assuming we have declared a EZUtility object as 'ezu' and have a mitig

        >>> ezu.utility(m)
        array([ 9.83391921])
        >>> utility_tree, cons_tree, cost_tree, ce_tree = ezu.utility(m, retur

        """
        utility_tree = BigStorageTree(subinterval_len=self.period_len, decision_
        cons_tree = BigStorageTree(subinterval_len=self.period_len, decision_tim
        ce_tree = BigStorageTree(subinterval_len=self.period_len, decision_times
        cost_tree = SmallStorageTree(decision_times=self.decision_times)
```

```
        self._end_period_utility(m, utility_tree, cons_tree, cost_tree)
        it = self._utility_generator(m, utility_tree, cons_tree, cost_tree, ce_t
        for u, period in it:
                utility_tree.set_value(period, u)

        if return_trees:
                return utility_tree, cons_tree, cost_tree, ce_tree
        return utility_tree[0]
```

_period_marginal_utility: a private function which returns the marginal utility for each node in a period.
Inputs:

- **prev_mu_0**: (ndarray) the number of it's decedent's node
- **prev_mu_1**: (ndarray) the number of it's decedent's node (always the same as **prev_mu_0** if we are dealing with one tree structure)
- **m**: (ndarray) mitigation level on each node
- **period**: the period that we are calculating
- **utility_tree**: (BigStorageTree Object): the information of it's children's utility
- **cons_tree**: (BigStorageTree Object): the information of it's children's consumption
- **ce_tree**: (SmallStorageTree Object): the information of it's children's ce

Outputs:

- **mu_0**: (float) the marginal utility with respect to consumption on this period
- **mu_1**: (float) the marginal utility with respect to the potential consumption on next period.
- **mu_2**: (float) the marginal utility w.r.t. the inter-decision-periods.

```
    def _period_marginal_utility(self, prev_mu_0, prev_mu_1, m, period, utility_tree
        """Marginal utility for each node in a period."""
        damage_period = utility_tree.between_decision_times(period)
        mu_0 = self._mu_0(cons_tree[period], ce_tree[period])

        prev_ce = ce_tree.get_next_period_array(period)
        prev_cons = cons_tree.get_next_period_array(period)
        if utility_tree.is_information_period(period):
                probs = self.tree.get_probs_in_period(damage_period+1)
                up_prob = np.array([probs[i]/(probs[i]+probs[i+1]) for i in rang
                down_prob = 1.0 - up_prob

                up_cons = prev_cons[::2]
```

```
                down_cons = prev_cons[1::2]
                up_ce = prev_ce[::2]
                down_ce = prev_ce[1::2]

                mu_1 = self._mu_1(cons_tree[period], up_prob, up_cons, down_cons
                mu_2 = self._mu_1(cons_tree[period], down_prob, down_cons, up_co
                return mu_0, mu_1, mu_2
            else:
                mu_1 = self._mu_2(cons_tree[period], prev_cons, prev_ce)
                return mu_0, mu_1, None
```

adjusted_utility: Calculating adjusted utility for sensitivity analysis using the marginal utility got from above functions. Used to

- Find the price of a bond that creates equal utility at time 0 as adding 'payment' to the value of consumption in the final period. (**find_ir** in analysis.py)
- Find the price of a bond that creates equal utility at time 0 as adding 'payment' to the value of consumption before the final period i.e. period T-1. (**find_term_structure** in analysis.py)
- Used to find a value for consumption that equalizes utility at time 0 in two different solutions. (**find_bec** in analysis.py)

Inputs:

- **m** : (ndarray) array of mitigations
- **node_cons_eps** : ('SmallStorageTree', optional) increases in consumption per node
- **period_cons_eps** : (ndarray, optional) array of increases in consumption per period
- **final_cons_eps** : (float, optional) value to increase the final utilities by
- **first_period_consadj** : (float, optional) value to increase consumption at period 0 by
- **return_trees** : (bool, optional) True if method should return trees calculated in producing the utility

Outputs:

- (ndarray or tuple) tuple of 'BaseStorageTree' including modified consumption tree, cost tree, utility tree and ce tree, if return_trees. Else ndarray with utility at period 0

```
    def adjusted_utility(self, m, period_cons_eps=None, node_cons_eps=None, final_co
                                                first_period_consadj=0.0, return_trees=
        """

        Examples
        ---------
        Assuming we have declared a EZUtility object as 'ezu' and have a mitig
```

```
>>> ezu.adjusted_utility(m, final_cons_eps=0.1)
array([ 9.83424045])
>>> utility_tree, cons_tree, cost_tree, ce_tree = ezu.adjusted_utility

>>> arr = np.zeros(int(ezu.decision_times[-1]/ezu.period_len) + 1)
>>> arr[-1] = 0.1
>>> ezu.adjusted_utility(m, period_cons_eps=arr)
array([ 9.83424045])

>>> bst = BigStorageTree(5.0, [0, 15, 45, 85, 185, 285, 385])
>>> bst.set_value(bst.last_period, np.repeat(0.01, len(bst.last)))
>>> ezu.adjusted_utility(m, node_cons_eps=bst)
array([ 9.83391921])

The last example differs from the rest in that the last values of the
used. Hence if you want to update the last period consumption, use one

>>> ezu.adjusted_utility(m, first_period_consadj=0.01)
array([ 9.84518772])

"""
utility_tree = BigStorageTree(subinterval_len=self.period_len, decision_
cons_tree = BigStorageTree(subinterval_len=self.period_len, decision_tim
ce_tree = BigStorageTree(subinterval_len=self.period_len, decision_times
cost_tree = SmallStorageTree(decision_times=self.decision_times)

periods = utility_tree.periods[::-1]
if period_cons_eps is None:
        period_cons_eps = np.zeros(len(periods))
if node_cons_eps is None:
        node_cons_eps = BigStorageTree(subinterval_len=self.period_len,

self._end_period_utility(m, utility_tree, cons_tree, cost_tree)

it = self._utility_generator(m, utility_tree, cons_tree, cost_tree, ce_t
i = len(utility_tree)-2
for u, period in it:
        if period == periods[1]:
                mu_0 = (1.0-self.b) * (u/cons_tree[period])**(1.0-self.r
                next_term = self.b * (1.0-self.b) / (1.0-self.b*self.gro
                mu_1 = (u**(1.0-self.r)) * next_term * (cons_tree.last**
                u += (final_cons_eps+period_cons_eps[-1]+node_cons_eps.l
                u +=  (period_cons_eps[i]+node_cons_eps.tree[period]) *
                utility_tree.set_value(period, u)
        else:
```

```
                        mu_0, m_1, m_2 = self._period_marginal_utility(mu_0, mu_
                        u +=  (period_cons_eps[i] + node_cons_eps.tree[period])*
                        utility_tree.set_value(period, u)
                i -= 1

        if return_trees:
                return utility_tree, cons_tree, cost_tree, ce_tree
        return utility_tree.tree[0]
```

marginal_utility: return the marginal utility trees got by function **_mu_i**(where $i = 0, 1, 2$) and function **_period_marginal_utility Inputs**:

- **m** : (ndarray) array of mitigations
- **utility_tree**: (BigStorageTree Object) place to store the calculated utility
- **cons_tree**: (BigStorageTree Object) place to store the calculated consumption
- **ce_tree**: (BigStorageTree Object) place to store the calculated modified ce-term

Outputs:

- **mu_tree_0**: (BigStorageTree Object) place to store the calculated mu_0 on each node.
- **mu_tree_1**: (BigStorageTree Object) place to store the calculated mu_1 on each node.
- **mu_tree_2**: (BigStorageTree Object) place to store the calculated mu_2 on each node. ce-term

```
    def marginal_utility(self, m, utility_tree, cons_tree, cost_tree, ce_tree):
        """Calculating marginal utility for sensitivity analysis, e.g. in the

        Parameters
        ----------
        m : ndarray
                array of mitigations
        utility_tree : `BigStorageTree` object
                utility values from using mitigation `m`
        cons_tree : `BigStorageTree` object
                consumption values from using mitigation `m`
        cost_tree : `SmallStorageTree` object
                cost values from using mitigation `m`
        ce_tree : `BigStorageTree` object
                certain equivalence values from using mitigation `m`

        Returns
        -------
        tuple
                marginal utility tree
```

```
        Examples
        --------
        Assuming we have declared a EZUtility object as 'ezu' and have a mitig
        >>>
        >>> utility_tree, cons_tree, cost_tree, ce_tree = ezu.utility(m, retur
        >>> mu_0_tree, mu_1_tree, mu_2_tree = ezu.marginal_utility(m, utility_
        >>> mu_0_tree[0] # value at period 0
        array([ 0.33001256])
        >>> mu_1_tree[0] # value at period 0
        array([ 0.15691619])
        >>> mu_2_tree[0] # value at period 0
        array([ 0.13948175])

        """
        mu_tree_0 = BigStorageTree(subinterval_len=self.period_len, decision_tim
        mu_tree_1 = BigStorageTree(subinterval_len=self.period_len, decision_tim
        mu_tree_2 = SmallStorageTree(decision_times=self.decision_times)

        self._end_period_marginal_utility(mu_tree_0, mu_tree_1, ce_tree, utility
        periods = utility_tree.periods[::-1]

        for period in periods[2:]:
                mu_0, mu_1, mu_2 = self._period_marginal_utility(mu_tree_0.get_n
                        mu_tree_1.get_next_period_array(period), m, period, util
                mu_tree_0.set_value(period, mu_0)
                mu_tree_1.set_value(period, mu_1)
                if mu_2 is not None:
                        mu_tree_2.set_value(period, mu_2)

        return mu_tree_0, mu_tree_1, mu_tree_2
```

partial_grad: Calculate the ith element of the gradient vector $dU(m)$, i.e. the utility function with mitigation level array as variable. **Inputs**:

- **m** : (ndarray) array of mitigation levels.
- **i** : (int) index of the target element in the mitigation array.

Outputs:

- **grad** : (float) gradient for the i th element in the gradient vector, equals to $\frac{U(m)}{m_i}$

```
    def partial_grad(self, m, i, delta=1e-8):

        m_copy = m.copy()
        m_copy[i] -= delta
        minus_utility = self.utility(m_copy)
        m_copy[i] += 2*delta
```

```
plus_utility = self.utility(m_copy)
grad = (plus_utility-minus_utility) / (2*delta)
return grad
```