Documentation for Utility.py

Yili Yang

June 2017

1 Introduction

Our model use preference specification suggested by Epstein and Zin that allows for 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, and 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 for calibration across states of nature and time.
put (2) in to (1) we get:

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

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

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

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

with

$$c_0 = \bar{c_0} \cdot (1 - \kappa_0(x_0)) \tag{6}$$

$$c_t = \bar{c_t} \cdot (1 - D_t(X_t, \theta_t) - \kappa_t(x_t)) fort \in \{1, 2, \ldots, T-1\} \tag{7}$$

$$c_T = \bar{c_T} \cdot (1 - D_T(X_T, \theta_T)) \tag{8}$$

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:

$$X_t = \frac{\sum_{s=0}^{t} g_s \cdot x_s}{\sum_{s=0}^{t} g_s} \tag{9}$$

2 EZUtility Class

This is a class to calculate the EZ-utility.

2.1 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** : (float) subinterval length
- **eis** : (float, optional) $\sigma = 1/(1-\rho)$ elasticity of intertemporal substitution
- **ra** : (float, optional) $\gamma = 1 - \alpha$ risk-aversion
- **time_pref** : (float, optional) $(1-\beta)/\beta$ 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 generates the utility of each node as well as consumption, cost and certainty-equivalent.

2.2 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, years in the future where decisions will be made
- **cons_growth** : (float) attr from damage object, consumption growth
- **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 patameter β in the introductoin

2.3 Methods

_end_period_utility: private function to calculate the utility on the final stage.
Using the equation (5) to get he utility on each node within the final period, where c_T is from equation (8) given damage from simulation.

```
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) # a

        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 and marginal utility w.r.t the consumption next period on the final stage and the last stage before end denoted as $T-1$ when we get the full information.
The utility of the final stage can 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 defined by equation (2). The certainty-equivalent term in the final stage is

$$U_T - (1-\beta)c_T^{\rho} \tag{10}$$

The marginal utility w.r.t the consumption function mu_0^t is derived by following equation.

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

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

And the marginal utility at final stage mu_0^T is got by:

$$\rho U_t^{\rho-1}dU_T = \frac{(1-\beta)}{1-\beta(1+g)^{\rho}}{}^{\rho-1}_{T} \tag{13}$$

$$\Rightarrow dU_T = (1-\beta)*(\frac{u_{T-1}}{c_T})^{1-\rho}dT \tag{14}$$

For the marginal utility w.r.t the consumption next time. First, we consider the $T-1$ stage since the final stage don't have term "consumption next time". For this stage,

$$U_{T-1}^{\rho} = (1-\beta)c_{T-1}^{\rho} + \beta U_T^{\rho} \tag{15}$$

$$U_{T-1}^{\rho} = (1-\beta)c_{T-1}^{\rho} + \frac{\beta*(1-\beta)}{1-\beta(1+g)^{\rho}}c_T^{rho} \tag{16}$$

$$\Rightarrow\Rightarrow \rho U_{T-1}^{\rho-1} = \rho\frac{\beta*(1-\beta)}{1-\beta(1+g)^{\rho}}c_T^{rho-1} \tag{17}$$

```
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 ability weighted sum of next period utility over the partition reachable from the state.

$$E_t[U_{t+1}^{\alpha}]^{1/\alpha} = \sum_{state\in mathbbs}[(U_{t+1}^{\alpha}|state)\cdot\mathbf{p}(state)]^{1/\alpha} \tag{18}$$

where $\sim$ is a set of all the possible states

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

        return cert_equiv
```

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

- For decision time, consumption is $(the\ init\ consumption)^(1 + constant\ growth\ rate * years) * (1 - damage) * (1 - cost)$

- 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 nodes.

```python
def _utility_generator(self, m, utility_tree, cons_tree, cost_tree, ce_tree, con
        """Generator for calculating utility for each utility period besides th
        # 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: 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]
```

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

```
	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 consumption next

period.
Potential Bug: For starters, the sampling space need to satisfy certain condition (for example Radon-Nikodym Theorem) when we want to take derivatives of random variables, but the target's condition might not be good enough.
Secondly, even if we take derivatives of certain-equivalence term:$[\mu(\tilde{U}_{t+1})]$, the following code is wrong. for the stage right before final stage, we have proved that the formula is equation (15), and for a $U_t, t \in \{0, 1, 2, \ldots, T-2\}$ the next stage is either up or down, we denote the 'up' utility as U_{t_1} with probability p_1 and down as U_{t_2} with probability $p_2 = 1 - p_1$ according to equation (4) they can be write as:

$$U_{t_i} = [(1-\beta)c_{t_i}^{\rho} + ce_{t_i}]^{1/\rho} where\ i = 1, 2 \tag{19}$$

where the ce term is defined in the **_utility_generator** function as:

$$ce = \beta[\mu_t(\tilde{U}_{t+1})]^{\rho}] \tag{20}$$

put (19) in to (4):

$$U_t = [(1-\beta)c_t^{\rho} + \beta(U_{t_1}^{\alpha} * p_1 + U_{t_2}^{\alpha} * p_2)^{\frac{\rho}{\alpha}}]^{1/\rho} \tag{21}$$

to perform derivation on (21) 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

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

and

$$f'(U_{t_i}) = U'_{t_i} * f'(U_{t_i})\rho(1-\beta)c_{t_i}^{\rho-1} * \frac{1}{\rho}[(1-\beta)c_{t_i}^{\rho} + ce_{t_i}]^{1/\rho-1} * \alpha * [(1-\beta)c_{t_i}^{\rho} + ce_{t_i}]^{\alpha/\rho-1} * p_i where\ i = 1, 2 \tag{23}$$

then

$$U'_t = (\sum f'(U_{t_i})) \cdot (\sum f(U_{t_i}))^{\frac{\rho}{\alpha}-1} \cdot \frac{\rho}{\alpha}\beta * (1/\rho) * U_t^{\frac{\rho}{1/(\rho-1)}} \tag{24}$$

```
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
```

If we are dealing with the last period, there is no uncertainty w.r.t. the consumption next time, then the margin is only generated by the constant increase of the future margin.

```
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
```

_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: return the marginal utilities and also modify

```
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. 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. **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' 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** and function **_period_marginal_utility**

```
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

```
def partial_grad(self, m, i, delta=1e-8):
        """Calculate the ith element of the gradient vector.

        Parameters
        ----------
        m : ndarray
                array of mitigations
        i : int
                node to calculate partial grad for

        Returns
        -------
        float
                gradient element

        """
        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
```