Documentation for Analysis.py

Yili Yang

June 2017

1 Introduction

This code file is used to do sensitivity analysis about the EZ-climate model.

2 Methods

additional_ghg_emission: additional ghg caused by emission is the mitigation level now times the ghg level change w.r.t the emission in this period. **Inputs**:

- **m** : (ndarray) mitigation level on each node.
- **utility**: ('Utility' object) instance of utility class.

Outputs:
additional GHG emission on each node.

```
def additional_ghg_emission(m, utility):

        additional_emission = np.zeros(len(m))
        cache = set()
        for node in range(utility.tree.num_final_states, len(m)): # the number of final
                path = utility.tree.get_path(node)
                for i in range(len(path)):
                        if path[i] not in cache:
                                additional_emission[path[i]] = (1.0 - m[path[i]]) *  uti
                                cache.add(path[i])
        return additional_emission
```

store_trees: save the values in BaseStorageTree to a csv file using **write_columns** in storage tree class. Refer the documentation of storage tree to get the structure of saved csv.

```
def store_trees(prefix=None, start_year=2015, **kwargs):
        """Saves values of `BaseStorageTree` objects. The file is saved into the 'data
        in the current working directory. If there is no 'data' directory, one is crea
```

```
        Parameters
        ----------
        prefix : str, optional
                prefix to be added to file_name
        start_year : int, optional
                start year of analysis
        **kwargs
                arbitrary keyword arguments of `BaseStorageTree` objects

        """
        if prefix is None:
                prefix = ""
        for name, tree in kwargs.items():
                tree.write_columns(prefix + "trees", name, start_year)
```

delta_consumption: Calculate the changes in consumption and the mitigation cost component of consumption when increasing period 0 mitigation with 'delta_m'. **Inputs**:

- **m** : (ndarray) mitigation level on each node.
- **utility**: ('Utility' object) instance of utility class.
- **cons_tree**: ('BigStorageTree' object) place to save the modified consumption.
- **cost_tree**: ('SmallStorageTree' object) place to save the modified cost.
- **delta**: (float) value to increase period 0 mitigation by

Outputs:
returns a tuple contains:

- storage tree of changes in consumption per delta m
- ndarray of costs in first periods
- new utility at the start point

```
def delta_consumption(m, utility, cons_tree, cost_tree, delta_m):

        m_copy = m.copy()
        m_copy[0] += delta_m

        new_utility_tree, new_cons_tree, new_cost_tree, new_ce_tree = utility.utility(m_

        for period in new_cons_tree.periods:
                new_cons_tree.tree[period] = (new_cons_tree.tree[period]-cons_tree.tree[

        first_period_intervals = new_cons_tree.first_period_intervals
        cost_array = np.zeros((first_period_intervals, 2))
```

```
    for i in range(first_period_intervals):
            potential_consumption = (1.0 + utility.cons_growth)**(new_cons_tree.subi
            cost_array[i, 0] = potential_consumption * cost_tree[0]
            cost_array[i, 1] = (potential_consumption * new_cost_tree[0] - cost_arra

    return new_cons_tree, cost_array, new_utility_tree[0]
```

constraint_first_period: Calculate the changes in consumption, the mitigation cost component of consumption, and new mitigation values when constraining the first period mitigation to a given 'first_node'. **Inputs**:

- **m**: (ndarray) mitigation level on each node
- **utility**: ('Utility' object) object of utility class
- **first_node**: (float) value to constrain first period to

Outputs:

- **new mitigation array**: (ndarray) the modified mitigation level on each node

```
def constraint_first_period(utility, first_node, m_size):
    """Calculate the changes in consumption, the mitigation cost component of cons
    and new mitigation values when constraining the first period mitigation to `fi


    """
    #fix the first period
    fixed_values = np.array([first_node])
    fixed_indicies = np.array([0])
    ga_model = GeneticAlgorithm(pop_amount=150, num_generations=100, cx_prob=0.8, mu
                                                                num_feature=m_size, util
                                                                fixed_indicies=fixed_ind

    gs_model = GradientSearch(var_nums=m_size, utility=utility, accuracy=1e-7,
                                                        iterations=250, fixed_values=f
                                                        print_progress=True)
    #run opt again
    final_pop, fitness = ga_model.run()
    sort_pop = final_pop[np.argsort(fitness)][::-1]
    new_m, new_utility = gs_model.run(initial_point_list=sort_pop, topk=1)
    return new_m
```

All of the four functions below are finding a point between interval $[a, b] \in \mathbb{R}$ where the first variable equals the second using brentq method from scipy package. Brentq method is using the classic Brent (1973) method to find a zero of the function f on the sign changing interval [a , b]. And here, the function f is the difference between the first and second variable. And therefore, the returning root the the difference function is making the first and second varible equal.

find_ir: 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. The purpose of this function is to find the interest rate embedded in the 'EZUtility' model. The first variable here is the utility with the final payment and the second variable is the utility with the initial payment.
Inputs:

- **m**:(ndarray) mitigation level at each node
- **utility**: (Utility object) instance of utility object
- **a,b** : (float,optional) lower and upper bound of the search area, [a,b] need to contain zero.

Outputs:
the price of the targeting bond.

```
def find_ir(m, utility, payment, a=0.0, b=1.0):
        """Find the price of a bond that creates equal utility at time 0 as adding `pa
        consumption in the final period. The purpose of this function is to find the i
        embedded in the `EZUtility` model.


        Note
        ----
        requires the 'scipy' package

        """

        def min_func(price):
                utility_with_final_payment = utility.adjusted_utility(m, final_cons_eps=
                first_period_eps = payment * price
                utility_with_initial_payment = utility.adjusted_utility(m, first_period_
                return utility_with_final_payment - utility_with_initial_payment

        return brentq(min_func, a, b)
```

find_term_structure: Find the price of a bond that creates equal utility at time 0 as adding 'payment' to the value of consumption just before the final period. The purpose of this function is to find the interest rate. embedded in the 'EZUtility' model. The first variable here is the utility with fix payment just before the final period and the second variable is the utility with the initial payment.
Inputs:

- **m**:(ndarray) mitigation level at each node
- **utility**: (Utility object) instance of utility object
- payment: (float) value added to consumption in the final period

- **a,b** : (float,optional) lower and upper bound of the search area, [a,b] need to contain zero.

Outputs:
the price of the targeting bond.

```
def find_term_structure(m, utility, payment, a=0.0, b=1.5):
        """Find the price of a bond that creates equal utility at time 0 as adding `pa
        consumption just before the final period. The purpose of this function is to f
        embedded in the `EZUtility` model.

        Note
        ----
        requires the 'scipy' package

        """

        def min_func(price):
                period_cons_eps = np.zeros(int(utility.decision_times[-1]/utility.period
                period_cons_eps[-2] = payment
                utility_with_payment = utility.adjusted_utility(m, period_cons_eps=perio
                first_period_eps = payment * price
                utility_with_initial_payment = utility.adjusted_utility(m, first_period_
                return  utility_with_payment - utility_with_initial_payment

        return brentq(min_func, a, b)
```

find_bec: Used to find a value for consumption that equalizes utility at time 0 in two different solutions. The first variable here is utility with one init consumption and the second variable is the utility with another init consumption
Inputs:

- **m**:(ndarray) mitigation level at each node
- **utility**: (Utility object) instance of utility object
- constraint_cost: (float) utility cost of constraining period 0 to zero
- **a,b** : (float,optional) lower and upper bound of the search area, [a,b] need to contain zero.

Outputs:
the price of the targeting consumption.

```
def find_bec(m, utility, constraint_cost, a=-0.1, b=1.5):
        """Used to find a value for consumption that equalizes utility at time 0 in tw

        Note
        ----
```

```
        requires the 'scipy' package

        """

        def min_func(delta_con):
                base_utility = utility.utility(m)
                new_utility = utility.adjusted_utility(m, first_period_consadj=delta_con
                print(base_utility, new_utility, constraint_cost)
                return new_utility - base_utility - constraint_cost

        return brentq(min_func, a, b)
```

perpetuity_yield: Find the yield of a perpetuity starting at year ‘start_date‘. The first variable here is the final price and the second is

$$[\frac{100}{(perp\_yield+100)^{start\_date}}(perp\_yield+100)]/(perp\_yield) \quad (1)$$

Inputs:

- **price**:(float) price of bond ending at ‘start_date‘
- **start_date**: (int) start year of perpetuity
- **a,b** : (float,optional) lower and upper bound of the search area, [a,b] need to contain zero.

Outputs:
the yield of a perpetuity.

```
def perpetuity_yield(price, start_date, a=0.1, b=10.0):
        """Find the yield of a perpetuity starting at year `start_date`.


        Note
        ----
        requires the 'scipy' package

        """

        def min_func(perp_yield):
                return price - (100. / (perp_yield+100.))**start_date * (perp_yield + 10

        return brentq(min_func, a, b)
```

2.1 Climate Output Class

Calculate and save output from the EZ-Climate model

2.1.1 Inputs and Outputs

Inputs:

- **utility** : (Utility object) object of utility class

Outputs: Calculated values based on optimal mitigation. For every **node** the function calculates and saves:

- average mitigation
- average emission
- GHGs level
- SCC

For every **period** the function also calculates and saves:

- expected SCC/price
- expected mitigation
- expected emission

2.1.2 Attributes

- **utility** : ('Utility' object) object of utility class
- **prices** : ndarray SCC prices for each node
- **ave_mitigations** : (ndarray) average mitigations
- **ave_emissions** : (ndarray) average emissions
- **expected_period_price** : (ndarray) expected SCC for the period
- **expected_period_mitigation** : (ndarray) expected mitigation for the period
- **expected_period_emissions** : (ndarray) expected emission for the period

```
def __init__(self, utility):
        self.utility = utility
        self.prices = None
        self.ave_mitigations = None
        self.ave_emissions = None
        self.expected_period_price = None
        self.expected_period_mitigation = None
        self.expected_period_emissions = None
        self.ghg_levels = None

def calculate_output(self, m):
```

```
"""Calculated values based on optimal mitigation. For every **node** the funct

        * average mitigation
        * average emission
        * GHG level
        * SCC

as attributes.

For every **period** the function also calculates and saves

        * expected SCC/price
        * expected mitigation
        * expected emission

as attributes.

Parameters
----------
m : ndarray or list
        array of mitigation

"""

bau = self.utility.damage.bau
tree = self.utility.tree
periods = tree.num_periods

self.prices = np.zeros(len(m))
self.ave_mitigations = np.zeros(len(m))
self.ave_emissions = np.zeros(len(m))
self.expected_period_price = np.zeros(periods)
self.expected_period_mitigation = np.zeros(periods)
self.expected_period_emissions = np.zeros(periods)
additional_emissions = additional_ghg_emission(m, self.utility)
self.ghg_levels = self.utility.damage.ghg_level(m)

for period in range(0, periods):
        years = tree.decision_times[period]
        period_years = tree.decision_times[period+1] - tree.decision_times[perio
        nodes = tree.get_nodes_in_period(period)
        num_nodes_period = 1 + nodes[1] - nodes[0]
        period_lens = tree.decision_times[:period+1]
```

```
            for node in range(nodes[0], nodes[1]+1):
                path = np.array(tree.get_path(node, period))
                new_m = m[path]
                mean_mitigation = np.dot(new_m, period_lens) / years
                price = self.utility.cost.price(years, m[node], mean_mitigation)
                self.prices[node] = price
                self.ave_mitigations[node] = self.utility.damage.average_mitigat
                self.ave_emissions[node] = additional_emissions[node] / (period_

            probs = tree.get_probs_in_period(period)
            self.expected_period_price[period] = np.dot(self.prices[nodes[0]:nodes[1
            self.expected_period_mitigation[period] = np.dot(self.ave_mitigations[no
            self.expected_period_emissions[period] = np.dot(self.ave_emissions[nodes

    def save_output(self, m, prefix=None):
        """Function to save calculated values in `calculate_output` in the file `prefi
        in the 'data' directory in the current working directory.

        The function also saves the values calculated in the utility function in the f
        `prefix` + 'tree' in the 'data' directory in the current working directory.

        If there is no 'data' directory, one is created.

        Parameters
        ----------
        m : ndarray or list
            array of mitigation
        prefix : str, optional
            prefix to be added to file_name

        """
        utility_tree, cons_tree, cost_tree, ce_tree = self.utility.utility(m, return_tre

        if prefix is not None:
            prefix += "_"
        else:
            prefix = ""

        write_columns_csv([m, self.prices, self.ave_mitigations, self.ave_emissions, sel
                          prefix+"node_period_output", ["Node", "Mitigation", "Prices",
                          "Average Emission", "GHG Level"], [range(len(m))])

        append_to_existing([self.expected_period_price, self.expected_period_mitigation,
                                           prefix+"node_period_output", header=["Pe
                                           "Expected Emission"], index=[range(self.
```

```
        store_trees(prefix=prefix, Utility=utility_tree, Consumption=cons_tree,
                    Cost=cost_tree, CertainEquivalence=ce_tree)
```

2.2 Risk Decomposition Class

new risk decomposition method, need the new paper to document it.

```
class RiskDecomposition(object):
        """Calculate and save analysis of output from the EZ-Climate model.

        Parameters
        ----------
        utility : `Utility` object
                object of utility class

        Attributes
        ----------
        utility : `Utility` object
                object of utility class
        sdf_tree : `BaseStorageTree` object
                SDF for each node
        expected_damages : ndarray
                expected damages in each period
        risk_premium : ndarray
                risk premium in each period
        expected_sdf : ndarray
                expected SDF in each period
        cross_sdf_damages : ndarray
                cross term between the SDF and damages
        discounted_expected_damages : ndarray
                expected discounted damages for each period
        net_discount_damages : ndarray
                net discount damage, i.e. when cost is also accounted for
        cov_term : ndarray
                covariance between SDF and damages

        """

        def __init__(self, utility):
                self.utility = utility
                self.sdf_tree = BigStorageTree(utility.period_len, utility.decision_time
                self.sdf_tree.set_value(0, np.array([1.0]))
```

```
        n = len(self.sdf_tree)
        self.expected_damages = np.zeros(n)
        self.risk_premiums = np.zeros(n)
        self.expected_sdf = np.zeros(n)
        self.cross_sdf_damages = np.zeros(n)
        self.discounted_expected_damages = np.zeros(n)
        self.net_discount_damages = np.zeros(n)
        self.cov_term = np.zeros(n)

        self.expected_sdf[0] = 1.0


    def sensitivity_analysis(self, m):
        """Calculate sensitivity analysis based on the optimal mitigation. For
        periods given by the utility calculations, the function calculates and

                * discount prices
                * net expected damages
                * expected damages
                * discounted expected damages
                * risk premium
                * cross SDF & damages
                * covariance between SDF and damages

        as attributes.

        Parameters
        ----------
        m : ndarray or list
                array of mitigation
        utility : `Utility` object
                object of utility class
        prefix : str, optional
                prefix to be added to file_name

        """

        utility_tree, cons_tree, cost_tree, ce_tree = self.utility.utility(m, re
        cost_sum = 0
        # Calculate the changes in consumption and the mitigation cost compone
        self.delta_cons_tree, self.delta_cost_array, delta_utility = delta_consu
        # Calculate the marginal utilities
        mu_0, mu_1, mu_2 = self.utility.marginal_utility(m, utility_tree, cons_t
        sub_len = self.sdf_tree.subinterval_len
        i = 1
```

```
        # for every period in sdf_tree (except the init point),
        for period in self.sdf_tree.periods[1:]:
            node_period = self.sdf_tree.decision_interval(period)
            period_probs = self.utility.tree.get_probs_in_period(node_period
            expected_damage = np.dot(self.delta_cons_tree[period], period_pr
            self.expected_damages[i] = expected_damage # calculate the expec

            if self.sdf_tree.is_information_period(period-self.sdf_tree.subi
                total_probs = period_probs[::2] + period_probs[1::2]
                mu_temp = np.zeros(2*len(mu_1[period-sub_len]))
                mu_temp[::2] = mu_1[period-sub_len]
                mu_temp[1::2] = mu_2[period-sub_len]
                sdf = (np.repeat(total_probs, 2) / period_probs) * (mu_t
                period_sdf = np.repeat(self.sdf_tree.tree[period-sub_len
            else:
                sdf = mu_1[period-sub_len]/mu_0[period-sub_len]
                period_sdf = self.sdf_tree[period-sub_len]*sdf

            self.expected_sdf[i] = np.dot(period_sdf, period_probs)
            self.cross_sdf_damages[i] = np.dot(period_sdf, self.delta_cons_t
            self.cov_term[i] = self.cross_sdf_damages[i] - self.expected_sdf

            self.sdf_tree.set_value(period, period_sdf)

            if i < len(self.delta_cost_array):
                self.net_discount_damages[i] = -(expected_damage + self.
                cost_sum += -self.delta_cost_array[i, 1] * self.expected
            else:
                self.net_discount_damages[i] = -expected_damage * self.e

            self.risk_premiums[i] = -self.cov_term[i]/self.delta_cons_tree[0
            self.discounted_expected_damages[i] = -expected_damage * self.ex
            i += 1

    def save_output(self, m, prefix=None):
        """Save attributes calculated in `sensitivity_analysis` into the file p
        in the `data` directory in the current working directory.

        Furthermore, the perpetuity yield, the discount factor for the last pe
        expected damage and risk premium for the first period is calculated an
        prefix + `tree` in the `data` directory in the current working directo
        one is created.

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

```
            m : ndarray or list
                    array of mitigation
            prefix : str, optional
                    prefix to be added to file_name

            """
            end_price = find_term_structure(m, self.utility, 0.01)
            perp_yield = perpetuity_yield(end_price, self.sdf_tree.periods[-2])

            damage_scale = self.utility.cost.price(0, m[0], 0) / (self.net_discount_
            scaled_discounted_ed = self.net_discount_damages * damage_scale
            scaled_risk_premiums = self.risk_premiums * damage_scale

            if prefix is not None:
                    prefix += "_"
            else:
                    prefix = ""

            write_columns_csv([self.expected_sdf, self.net_discount_damages, self.ex
                               self.cross_sdf_damages, self.discounted_expected_
                               scaled_discounted_ed, scaled_risk_premiums], pref
                                            ["Year", "Discount Prices", "Net Expe
                                             "Cross SDF & Damages", "Discounted E
                                             "Scaled Risk Premiums"], [self.sdf_t

            append_to_existing([[end_price], [perp_yield], [scaled_discounted_ed.sum
                                [self.utility.cost.price(0, m[0], 0)]], prefix+"
                                header=["Zero Bound Price", "Perp Yield", "Expec
                                                "SCC"], start_char='\n')

            store_trees(prefix=prefix, SDF=self.sdf_tree, DeltaConsumption=self.delt
```

2.3 Constraint Analysis Class

Analysis of adding constraint to the original model.

2.3.1 Input

- **utility**: ('utility' Object) the utility without change
- **const_value**: a float that represents the scale value to constrain the range of change.

2.3.2 Output

- **con_cost**: constraint cost = optimum cost - cost with constraint on first period
- **Delta Consumption**: Delta Consumption = consumption that equalizes utility at time 0 in two different solutions
- **Delta Consumption $b**: Consumption in Billions = Delta Consumption × consumption per ton constant / emit level on init point
- **Delta Emission Gton**: = optimum emit level on the init point
- **Deadweight Cost**: deadweight cost = Delta Consumption × consumption per ton constant / optimum mitigation level on init point
- **Marginal Impact Utility**: = change of utility when add 0.01 on the init consumption
- **Marginal Benefit Emissions Reduction**: = change when change mitigation level on init point by 0.01 / change when change consumption level on init point by 0.01 × consumption per ton constant
- **Marginal Cost Emission Reduction**: = change on price when add constraint on the first period

where consumption per ton constant is equal to $\frac{cons\_at\_0}{emit\_at\_0}$

```
class ConstraintAnalysis(object):
        def __init__(self, run_name, utility, const_value, opt_m=None):
                self.run_name = run_name
                self.utility = utility
                self.cfp_m = constraint_first_period(utility, const_value, utility.tree.
                self.opt_m = opt_m
                if self.opt_m is None:
                        self.opt_m = self._get_optimal_m()

                self.con_cost = self._constraint_cost()
                self.delta_u = self._first_period_delta_udiff()

                self.delta_c = self._delta_consumption()
                self.delta_c_billions = self.delta_c * self.utility.cost.cons_per_ton \
                                                                        * self.utility.damage.ba
                self.delta_emission_gton = self.opt_m[0]*self.utility.damage.bau.emit_le
                self.deadweight = self.delta_c*self.utility.cost.cons_per_ton / self.opt

                # adjusted benefit when +0.01 to the mitigation level at time zero
                self.delta_u2 = self._first_period_delta_udiff2()
                self.marginal_benefit = (self.delta_u2 / self.delta_u) * self.utility.co
                self.marginal_cost = self.utility.cost.price(0, self.cfp_m[0], 0)
```

```python
def _get_optimal_m(self):
        try:
                header, index, data = import_csv(self.run_name+"_node_period_out
        except:
                print("No such file for the optimal mitigation..")
        return data[:, 0]

def _constraint_cost(self):
        opt_u = self.utility.utility(self.opt_m)
        cfp_u = self.utility.utility(self.cfp_m)
        return opt_u - cfp_u

def _delta_consumption(self):
        return find_bec(self.cfp_m, self.utility, self.con_cost) # value for co

def _first_period_delta_udiff(self):
        u_given_delta_con = self.utility.adjusted_utility(self.cfp_m, first_peri
        cfp_u = self.utility.utility(self.cfp_m)
        return u_given_delta_con - cfp_u

def _first_period_delta_udiff2(self):
        m = self.cfp_m.copy()
        m[0] += 0.01 # adjusted with a fixed number
        u = self.utility.utility(m)
        cfp_u = self.utility.utility(self.cfp_m)
        return u - cfp_u

def save_output(self, prefix=None):
        if prefix is not None:
                prefix += "_"
        else:
                prefix = ""

        write_columns_csv([self.con_cost, [self.delta_c], [self.delta_c_billions
                                   [self.deadweight], self.delta_u, self
                                   prefix + self.run_name + "_constraint
                                  ["Constraint Cost", "Delta Consumption
                                   "Delta Emission Gton", "Deadweight Co
                                   "Marginal Benefit Emissions Reduction
```