Documentation for Damage_simulation.py

Yili Yang

June 2017

1 Introduction

The damage function simulation is a key input into the pricing engine. Damages are represented in arrays of dimension $n \times p$, where n, p represent numbers of states and periods respectively. The arrays are created by Monte Carlo simulation. Each array specifies for each state and time period a damage coefficient.
Up to a point, the Monte Carlo follows Pindyck (2012) 'Uncertain Outcomes and Climate Change Policy':

1. There is a gamma distribution for temperature
2. There is a gamma distribution for economic impact (conditional on temperature)

However, in addition, this program adds a probability of a tipping point (conditional on temperature). This probability is a decreasing function of the parameter 'peak_temp', conditional on a tipping point. Damage itself is a decreasing function of the parameter 'disaster_tail'.

For the base case, the damage is simulated from the gamma distribution given by pindyck using 3 base scenarios [450,650,1000] for GHG level with full, half and no mitigation to simulate the temperature and then use the temperature to calculate economic impact and finally calculates the damage. Then get the damage for each period using a method **_damage_function_node**in damage.py

2 Damage Simulation Class

It is a class with a main function **simulate** which returns the simulated damage given a simulation method. The following methods are supported:

1. Pindyck displace gamma
2. Wagner-Weitzman normal
3. Roe-Baker
4. user-defined normal

5. user-defined gamma

2.1 Inputs and Outputs

Inputs:

- **tree** : (‘TreeModel‘ object) tree structure that been used
- **ghg_levels** : (ndarray or list) end GHG level for each scenarios.
- **peak_temp** : (float) tipping point parameter
- **disaster_tail** : (float) curvature of tipping point
- **tip_on** : (bool) flag that turns tipping points on or off
- **temp_map** : (int) 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) if temp_map is either 3 or 4, user needs to define the distribution parameters
- **maxh** : (float) time parameter from Pindyck which indicates the time it takes for temp to get half way to its max value for a given level of ghg
- **cons_growth** : (float) yearly growth in consumption, in our example, this number is 0.015.

Outputs:
The main output for this class is from function **simulate** which returns a 2-D array of damage indexed by $x = number\ of\ final\ states$ and $y = number\ of\ periods$. Notice that a child only have one parent and thus we can get a specific node given the final state and the period.

2.2 Attributes

- **tree** : (‘TreeModel‘ object) tree structure that has been used
- **ghg_levels** : (ndarray or list) end GHG level for each path
- **peak_temp** : (float) tipping point parameter
- **disaster_tail** : (float) curvature of tipping point

- **tip_on** : (bool) flag that turns tipping points on or off
- **temp_map** : (int) 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
- **dist_params** : (ndarray or list) if temp_map is either 3 or 4, user needs to define the distribution parameters
- **maxh** : (float) time parameter from Pindyck which indicates the time it takes for temp to get half way to its max value for a given level of ghg
- **cons_growth** : (float) yearly growth in consumption
- **d**: (2-d array) simulated damage for the final states and storage space for damage on a certain path. For example: if we have 2 final states and 2 periods. Then simulated d is like [[0.1,0],[0.2,0]] where 0.1,0.2 are the simulated values for the final states, and 0 is the storage space for damage for the init point if we are on path 1 or path 2. The damage on path will be calculated in damage and be recombined in damage.py

2.3 Methods

_gamma_array,**_normal_array**,**_uniform_array**: basic building functions to get random numbers of given dimension from gamma, normal and uniform distributions.

```
def _gamma_array(self, shape, rate, dimension):
    return np.random.gamma(shape, 1.0/rate, dimension)

def _normal_array(self, mean, stdev, dimension):
    return np.random.normal(mean, stdev, dimension)

def _uniform_array(self, dimension):
    return np.random.random(dimension)
```

_sort_array: sort a given 2-D array to arrange the entries in an increasing order with respect to their values of a given period. For example:

```
#given an array has 4 final states and 2 periods:
array([[2, 1],
       [4, 3],
       [5, 2],
       [7, 8]])
```

```
>>>array[array[:,1].argsort()] # same as _sort_array()
array([[2, 1],
       [5, 2],
       [4, 3],
       [7, 8]])

def _sort_array(self, array):
    return array[array[:, self.tree.num_periods-1].argsort()]
```

_normal_simulation: Draw random samples from normal distribution for mapping GHG to temperature given user defined parameter.

Inputs:

- **average**: (ndarray or list) : average temperature for each period
- **std** (ndarray or list) : standard deviation for each period

Outputs:

- 1-D array of $e^{simulated\ temperature}$

```
def _normal_simulation(self):
    """Draw random samples from normal distribution for mapping GHG to temperature
    user-defined distribution parameters.
    """
    assert self.temp_dist_params and len(self.temp_dist_params) == 2, "Normal distri

    ave, std = temp_dist_params
    n = len(ave)
    temperature = np.array([self._normal_array(ave[i],std[i], self.draws) for i in r
    return np.exp(temperature)
```

_gamma_simulation: Draw random samples from displaced gamma distribution for mapping GHG to temperature given user defined parameter.
Displaced gamma distribution is given by:

$$f(x; r, \lambda, \theta) = \frac{\lambda^r}{\Gamma(r)}(x-\theta)^{r-1}e^{-\lambda(x-\theta)}, x \geq \theta \tag{1}$$

where $\Gamma(r) = \int_0^\infty s^{r-1}e^{-s}ds$ is the gamma function.
However, we used $gamma(k, \theta) + displace$ to get the numerical result.
Inputs:

- **k**: (ndarray or list) : shape parameter for each period
- **theta** (ndarray or list) : scale parameter for each period
- **displace** (ndarray or list) : displacement parameter for each period

Outputs:

- 1-D array of simulated temperature

```
def _gamma_simulation(self):
    """Draw random samples from gamma distribution for mapping GHG to temperature 
    user-defined distribution parameters.
    """
    assert self.temp_dist_params and len(self.temp_dist_params) == 3, "Gamma distrib

    k, theta, displace = temp_dist_params
    n = len(k)
    return np.array([self._gamma_array(k[i], theta[i], self.draws)
                     + displace[i] for i in range(0, n)])
```

All the parameters below is lack of reference, there is an issue been created about this.
_pindyck_simulation: Draw random samples for mapping GHG to temperature based on Pindyck. It is drawing from a gamma distribution but with the parameter given by pindyck

```
def _pindyck_simulation(self):
    """Draw random samples for mapping GHG to temperature based on Pindyck. The `p
    is the shape parameter from Pyndyck damage function, `pindyck_impact_theta` th
    from Pyndyck damage function, and `pindyck_impact_displace` the displacement p
    damage function.
    """
    pindyck_temp_k = [2.81, 4.6134, 6.14]
    pindyck_temp_theta = [1.6667, 1.5974, 1.53139]
    pindyck_temp_displace = [-0.25, -0.5, -1.0]
    return np.array([self._gamma_array(pindyck_temp_k[i], pindyck_temp_theta[i], sel
                     + pindyck_temp_displace[i] for i in range(0, 3)])
```

_ww_simulation : Draw random samples for mapping GHG to temperature based on Wagner-Weitzman. It is a drawing from a normal distribution with the parameters given by Wagner-Weitzman.

```
def _ww_simulation(self):
    """Draw random samples for mapping GHG to temperature based on Wagner-Weitzman
    ww_temp_ave = [0.573, 1.148, 1.563]
    ww_temp_stddev = [0.462, 0.441, 0.432]
    temperature = np.array([self._normal_array(ww_temp_ave[i], ww_temp_stddev[i], se
                            for i in range(0, 3)])
    return np.exp(temperature)
```

_rb_simulation: It is drawing from a normal distribution with the parameters given by Roe-Baker.

```
def _rb_simulation(self):
    """Draw random samples for mapping GHG to temperature based on Roe-Baker."""
    rb_fbar = [0.75233, 0.844652, 0.858332]
    rb_sigf = [0.049921, 0.033055, 0.042408]
```

```
        rb_theta = [2.304627, 3.333599, 2.356967]
        temperature = np.array([self._normal_array(rb_fbar[i], rb_sigf[i], self.draws)
                          for i in range(0, 3)])
        return np.maximum(0.0, (1.0 / (1.0 - temperature)) - np.array(rb_theta)[:, np.ne
```

_pindyck_impact_simulation: It is drawing from a gamma distribution for the impact with the parameter given by pindyck

```
    def _pindyck_impact_simulation(self):
        """Pindyck gamma distribution mapping temperature into damages."""
        # get the gamma in loss function
        pindyck_impact_k=4.5
        pindyck_impact_theta=21341.0
        pindyck_impact_displace=-0.0000746,
        impact = self._gamma_array(pindyck_impact_k, pindyck_impact_theta, self.draws) +
                 pindyck_impact_displace
        return impact
```

lack of ref for parameters ends here

_disaster_simulation: Drawing random numbers from uniform distribution.

```
    def _disaster_simulation(self):
        """Simulating disaster random variable, allowing for a tipping point to occur
        with a given probability, leading to a disaster and a `disaster_tail` impact o
        """
        disaster = self._uniform_array((self.draws, self.tree.num_periods))
        return disaster
```

_disaster_cons_simulation: Generate TP_damage in the paper from a gamma distribution with parameters $\alpha = 1$ and $\beta = dosaster\_tail$.

```
    def _disaster_cons_simulation(self):
        """Simulates consumption conditional on disaster, based on the parameter disas
        #get the tp_damage in the article which is drawed from a gamma distri with alp
        disaster_cons = self._gamma_array(1.0, self.disaster_tail, self.draws)
        return disaster_cons
```

_interpolation_of_temp: for every temp in each period, modify it using the following equation:

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

where ΔT_{th} is half of the maximum temperature and th is the time to get the half of maximum temperature. It is equation (25) in the current paper.
For example, if it takes 50 years to reach half of the max temperature(assume max is 10), and now this has been 25 years, then the temperature now should be the fitted temperature times $2 \times 5 \times (1 - 0.5^{0.5}) \approx 0.586$ so that the temperature will have a non linear approximation.

```
    def _interpolation_of_temp(self, temperature):
            # for every temp in each period, modify it using a coff regards to the cur
```

```
        return temperature[:, np.newaxis] * 2.0 * (1.0 - 0.5**(self.tree.decision_times[
```

lack of reference about this formula **_economic_impact_of_temp**: calculate the economic impact of temperatures given temperature. This is used to take the temprature and calculate the resulting consumption which leads to damage level in the final period $damage = 1 - (consumption\ generated\ by\ this\ formula)(consumption\ with\ constant\ grow)$:

$$term_1 = \frac{-2 * simulated\_impact * maxh * temp(for\ each\ period)}{\log 0.5} \tag{3}$$

$$term_2 = con\_g - 2 * simulated\_impact * temp * time\_now \tag{4}$$

$$term_3 = \frac{2 * gamma * maxh * temp * 0.5^(time_{now}/maxh)}{\log(0.5)} \tag{5}$$

and the final consumption derived by the formula is $e^{term_1+term_2+term_3}$ **Inputs**:

- **temperature**:(ndarray) temperature generated by the above methods i.e. **_pindyck_simulation**....

Outputs:

- the consumption derived by the formula explained above.

```
    def _economic_impact_of_temp(self, temperature):
        """Economic impact of temperatures, Pindyck [2009]."""
        impact = self._pindyck_impact_simulation()
        term1 = -2.0 * impact[:, np.newaxis] * self.maxh * temperature[:,np.newaxis] / n
        term2 = (self.cons_growth - 2.0 * impact[:, np.newaxis] \
                * temperature[:, np.newaxis]) * self.tree.decision_times[1:] # con_g-2*g
        term3 = (2.0 * impact[:, np.newaxis] * self.maxh \
                * temperature[:, np.newaxis] * 0.5**(self.tree.decision_times[1:] / self
        return np.exp(term1 + term2 + term3)
```

_tipping_point_update: Determine whether a tipping point is hit, if so reduce consumption for all periods after this date. The step is as follows:

1. determine whether the tipping point is hit by comparing the probability of survival and a random number generated from uniform distribution, where the probability of survival is:
$$prob_{survival} = [1 - (\frac{tmp}{tmp\_scale})^2]^{\frac{period\_len}{peak\_interval}}$$
where tmp is the temperature change at time t(in years) denoted as $\Delta T(t)$, tmp_scale is $max[\Delta T(t), peakT]$
period_len is the length of decision interval that the time is in and period_interval is defaulted as 30 years in our base sample.

2. find unique final state and the periods that the disaster occurs and modify consumption after the point. (If a disaster happen more than once in a path, we only consider the influence of the first time.)

Inputs:

- **tmp** : smoothed temperature at a certain period
- **consump** : consumption at a certain period generated by _economic_impact_of_temp()
- **peak_temp_interval**: the time interval that is been used as benchmark, default to 30 but can be changed.

Outputs:

- Consumption after we determined whether tipping point is hit.

```
def _tipping_point_update(self, tmp, consump, peak_temp_interval=30.0):
    """Determine whether a tipping point has occurred, if so reduce consumption fo
    all periods after this date.
    """
    draws = tmp.shape[0]
    disaster = self._disaster_simulation()
    disaster_cons = self._disaster_cons_simulation()
    period_lengths = self.tree.decision_times[1:] - self.tree.decision_times[:-1]

    tmp_scale = np.maximum(self.peak_temp, tmp)
    ave_prob_of_survival = 1.0 - np.square(tmp / tmp_scale)
    prob_of_survival = ave_prob_of_survival**(period_lengths / peak_temp_interval) #
    # this part may be done better, this takes a long time to loop over
    # find unique row and the cols that the diaster occurs and modify consumption
    res = prob_of_survival < disaster
    rows, cols = np.nonzero(res)
    row, count = np.unique(rows, return_counts=True)
    first_occurance = zip(row, cols[np.insert(count.cumsum()[:-1],0,0)])
    for pos in first_occurance:
        consump[pos[0], pos[1]:] *= np.exp(-disaster_cons[pos[0]])
    return consump
```

_run_path: Calculate the distribution of damage for the final states.

Varibles:

- **tmp** : smoothed temperature at a certain period
- **consump** : consumption at a certain period generated by _economic_impact_of_temp()
- **peak_cons** : max consumption at a certain period generated by a constant growth rate: $exp(constant\ growth * time\ passed\ from\ the\ start\ point)$
- **damage** : 1.0 - (**consump** / **peak_cons**)
- **weights**: final_states_prob*number_of_ draws

To determine what state does the damage belong to, the code first sorts the simulated damage array by the simulated result then simply slice the damage array by the probability of a state occurs to classes. And then simply get the average within a class. (For example, there are two states "u" and "d" with probability 0.4 and 0.6. If we draw 10 times then the code first sort the array by the result number then take the mean of first 4 results (smallest 4) as the simulated damage for state "u" and the mean of latter 6 is the simulated damage for "d".)

Outputs:

- mean damage of the draws and return a 2-D array of damage illustrated in section 2.2.

```
def _run_path(self, temperature):
    """Calculate the distribution of damage for specific GHG-path. Implementation
    the temperature and economic impacts from Pindyck [2012] page 6.
    """
    # Remark
    # -------------
    # final states given periods can give us a specific state in that period since

    d = np.zeros((self.tree.num_final_states, self.tree.num_periods))
    tmp = self._interpolation_of_temp(temperature)
    consump = self._economic_impact_of_temp(temperature)
    peak_cons = np.exp(self.cons_growth*self.tree.decision_times[1:])

    # adding tipping points
    if self.tip_on:
        consump = self._tipping_point_update(tmp, consump)

    # sort based on outcome of simulation
    consump = self._sort_array(consump)
    damage = 1.0 - (consump / peak_cons)
    weights = self.tree.final_states_prob*(self.draws)
    weights = (weights.cumsum()).astype(int)

    d[0,] = damage[:weights[0], :].mean(axis=0)
    for n in range(1, self.tree.num_final_states):
        d[n,] = np.maximum(0.0, damage[weights[n-1]:weights[n], :].mean(axis=0))
    return d
```

simulate: main function of the class, multiprocessing **run_path** for a given method with simulated temperature.

```
def simulate(self, draws, write_to_file=True):
    """Create damage function values in 'p-period' version of the Summers - Zeckha

    Parameters
    ----------
    draws : int
        number of samples drawn in Monte Carlo simulation.
    write_to_file : bool, optional
```

```
            wheter to save simulated values

        Returns
        -------
        ndarray
            3D-array of simulated damages # it should be 2D : self.tree.num_final_stat

        Raises
        ------
        ValueError
            If temp_map is not in the interval 0-4.

        Note
        ----
        Uses the :mod:`~multiprocessing` package.

        """
        dnum = len(self.ghg_levels)
        self.draws = draws
        self.peak_cons = np.exp(self.cons_growth*self.tree.decision_times[1:])

        if self.temp_map == 0:
            temperature = self._pindyck_simulation()
        elif self.temp_map == 1:
            temperature = self._ww_simulation()
        elif self.temp_map == 2:
            temperature = self._rb_simulation()
        elif self.temp_map == 3:
            temperature = self._normal_simulation()
        elif self.temp_map == 4:
            temperature = self._gamma_simulation()
        else:
            raise ValueError("temp_map not in interval 0-4")

        pool = mp.Pool(processes=dnum)
        self.d = np.array(pool.map(self._run_path, temperature))

        if write_to_file:
            self._write_to_file()
        return self.d
```