RuinForest-Colin-Ghamnia-Grivet-simulations

December 11, 2023

1 Imports

[ ]:
```
import multiprocessing
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from typing import List, Callable, Optional
from scipy.stats import genpareto, poisson, expon, uniform
```

2 Functions defintion

[ ]:
```
def simulhpp(mean:float, T_max:int) -> List[float] :
  """
  Simulate a homogenous Poisson Process (hPP) with fixed T \\
  Input :
    mean : float : average number of events
    T_max : int : max time T
  Output :
    List[float] : times of occurence until T_max
  """
  T = expon.rvs(scale=1/mean, size=T_max)
  sT = np.cumsum(T)
  sT = sT[sT <= T_max]
  return np.round(sT)

  # Second method to simulate an hPP
  n = np.random.poisson(mean*T_max, 1)
  hpp = np.random.uniform(0, T_max, n)
  return np.round(np.sort(hpp))
```

[ ]:
```
def simulipp(lambda_:List[float], M:float) -> List[float] :
    """
    Simulate a inhomogenous Poisson Process (iPP) with fixed T \\
    Input :
```

```
        lambda_ : List[float] : intensity function values at times 1, ..., T_max
        M : float : upper-bound of lambda_
        T_max : int : max time T
    Output :
        List[float] : times of occurence until T_max
  """
    T_max = len(lambda_) - 1
    ti = simulhpp(M, T_max)
    n = len(ti)
    u = uniform.rvs(loc=0, scale=M, size=n)
    lambda_ti = [lambda_[int(t)] for t in ti]
    index_ti = u <= lambda_ti
    return sorted(ti[index_ti])
```

```
[ ]: def S(Ah:float, lambda_lowercase:float, xi:float, sigma:float, u:float) ->␣
     ↪float :
       """
       Compute damage cost of the year \\
       Input :
         Ah : float : normalizing constant that translates the climate hazard␣
     ↪conveyed by X_k into damage to R(t)
         lambda_lowercase : float : number of dry days
         xi : float : shape parameter of the GPD
         sigma : float : scale parameter of the GPD
         u : float : threshold of the GPD
       Output :
         float : damage cost of the year : Ah * sum(Xk) where Xk follow a pareto␣
     ↪distribution with paremter xi, sigma, u
       """
       Nt = poisson.rvs(lambda_lowercase)
       damage_by_days = genpareto.rvs(c=xi, loc=u, scale=sigma, size=Nt)
       return Ah * sum(damage_by_days)
```

```
[ ]: def p(p0:float, B:float, S_t_minus_one:float) -> float :
       """
       Compute net primary production (NPP) allocated to reserves of year t \\
       Input :
         p0 : float : optimum average yearly NPP of a population of trees allocated␣
     ↪to reserve
         B : float : memory factor of the damage function
         S_t_minus_one : damage cost of the previous year
       Output :
         float : NPP of year t : p0 - B*S_t_minus_one
       """
       return p0 - B*S_t_minus_one
```

[ ]:
```python
def R(R_t_minus_one:float, S_t : float, p_t : float, b:float, R_max:float) ->␣
↪float :
  """
  Compute the reserve (non-structural carbohydrates) that allows growth of tree␣
↪at the beginning of vegetative period for year t \\
  Input :
    R_t_minus_one : float : previous reserve
    S_t : float : damages of the current year
    p_t : float : NPP (tree Net Primary Production) of the current year
    b : float : fraction of previous resources devoted to growth
    R_max : float : maximal value of reserve that a tree can store
  Output :
    float : reserve of the year : max(min((1-b)*R_t_minus_one + p_t - S_t,␣
↪R_max), 0)
  """
  if R_t_minus_one == 0 : # Already ruined
    return 0              # So we can't produce anymore
  else :
    current_reserve = min((1-b)*R_t_minus_one + p_t - S_t, R_max)
    return max(current_reserve, 0)
```

[ ]:
```python
def compute_Ah_constant(B:float=0) -> float :
    """
    Compute Ah : the normalizing constant that translates the climate hazard␣
↪conveyed by Xk into damage to R(t) \\
    Input :
        B : float : memory parameter value (not use here beacuse Ah is constant)
    Output :
        float : 0.6
    """
    return 0.6
```

[ ]:
```python
def compute_Ah(B:float) -> float :
    """
    Compute Ah : the normalizing constant that translates the climate hazard␣
↪conveyed by Xk into damage to R(t) \\
    Input :
        B : float : memory parameter value
    Output :
        float : 1.2 / (1+B)
    """
    return 1.2 / (1+B)
```

[ ]:
```python
class SimuleTrajectories :
    def __init__(self, p0:float, b:float, R_max:float, sigma:float, xi:float, u:
↪float, compute_Ah:Callable,
```

```
                 lambda_lowercase:np.ndarray, lambda_uppercase:np.ndarray, B:
↪List[float], r0:float, s0:float) :
        """
        p0 : float : optimum average yearly NPP
        b : float : fraction of previous resources devoted to growth
        R_max : float : maximum amount of reserve that a tree can store
        sigma : float : scale parameter of the GPD
        xo : float : shape parameter of the GPD
        u : float : threshold parameter of the GPD
        compute_Ah : Callable : The function to compute Ah function of B
        lambda_lowercase : List[float] : average number of dry days in a year␣
↪of heatwave
        lambda_uppercase : float : average return period of heatwave
        B : List[float] : list of memory parameter values
        r0 : float : initial reserve capital
        s0 : float : damage of the year before the simulation start
        """
        self.n_year = None
        self.dataframes = []
        self.p0 = p0
        self.b = b
        self.R_max = R_max
        self.sigma = sigma
        self.xi = xi
        self.u = u
        self.compute_Ah = compute_Ah
        self.lambda_lowercase = lambda_lowercase
        self.lambda_uppercase = lambda_uppercase
        self.B = B
        self.r0 = r0
        self.s0 = s0
        self.nb_B = len(self.B)
        self.df = pd.DataFrame()
        self.df_variate = pd.DataFrame()
        self.n_sample = 0
        self.n_year = 0

    def __build_df(self) :
        return pd.DataFrame(columns=["B", "R", "S", "tau", "u", "sigma", "xi",␣
↪"lambda", "Lambda"])

    def simulate_trajectory(self, *args, **kwargs) :
        """
        Simulate reserve and damage over time
        """
        df = self.__build_df()
        for index_B, current_B in enumerate(self.B) :
```

```
            Ah = self.compute_Ah(current_B)
            year_of_heat_waves = simulipp(self.lambda_uppercase, max(self.
↪lambda_uppercase))
            S_t_minus_one = self.s0
            R_t_minus_one = self.r0
            R_list = []
            S_list = []
            tau = np.inf
            for year in range(1, self.n_year+1):
                if R_t_minus_one == 0 :
                    S_year = 0
                    R_year = 0
                else :
                    p_year = p(p0=self.p0, B=current_B,␣
↪S_t_minus_one=S_t_minus_one)
                    if year in year_of_heat_waves :
                        S_year = S(Ah=Ah, lambda_lowercase=self.
↪lambda_lowercase[year-1], xi=self.xi, sigma=self.sigma, u=self.u)
                    else:
                        S_year = 0
                    R_year = R(R_t_minus_one=R_t_minus_one, S_t=S_year,␣
↪p_t=p_year, b=self.b, R_max=self.R_max)
                R_list.append(R_year)
                S_list.append(S_year)
                S_t_minus_one = S_year
                R_t_minus_one = R_year
                if R_year == 0 and tau == np.inf :
                    tau = year
            df.loc[index_B] = [current_B, R_list, S_list, tau, self.u, self.
↪sigma, self.xi, self.lambda_lowercase[0], 1/self.lambda_uppercase[0]]
        return df

    def sample_trajectories(self, n_sample:int, t_max:int, variate:
↪Optional[str]=None) :
        """
        Do n_sample simulations of reserve and damage over time from 1 to t_max␣
↪and store it in self.df or in self.df_variate if variate != None \\
        Input :
            n_sample : int : number of simulation
            t_max : int : time to simulate per simulation
            variate : Optionnal[str] : indicate the parameter variating, None␣
↪if none
        """
        self.n_sample = n_sample
        self.n_year = t_max
        if variate :
```

```
            result_list = []
            for _ in range(n_sample) : # multiprocessing does not work with␣
↪variate... Don't know why...
                result_list.append(self.simulate_trajectory())
            df_variate = pd.concat(result_list)
            df_variate["variation"] = variate
            if self.df_variate.empty :
                self.df_variate = df_variate
            else :
                self.df_variate = pd.concat([self.df_variate, df_variate])
        else :
            p = multiprocessing.Pool(processes=multiprocessing.cpu_count())
            result_list = p.starmap(self.simulate_trajectory, [() for _ in␣
↪range(n_sample)])
            p.close()
            p.join()
            self.dataframes = result_list
            self.df = pd.concat(list(self.dataframes), ignore_index=True)
            self.df["R_mean"] = self.df["R"].apply(np.mean)

        del result_list[:]
        del self.dataframes[:]

    def sample_variate_params(self, n_sample:int, t_max:int, u_list:
↪List[float]=[], sigma_list:List[float]=[],
                              xi_list:List[float]=[], lambda_list:
↪List[float]=[], Lambda_list:List[float]=[]) :
        """
        Varies each parameter independently and store result in self.df_variate␣
↪\\
        Input :
            n_sample : int : number of simulation per parameter combination
            t_max : int : time to simulate per simulation
            u_list : List[float] : List of values for parameter u
            sigma_list : List[float] : List of values for parameter sigma
            xi_list : List[float] : List of values for parameter xi
            lambda_list : List[float] : List of values for parameter␣
↪lambda_lowercase (i.e. hPP only)
            Lambda_list : List[float] : List of values for parameter␣
↪lambda_uppercase (i.e. hPP only)
        """
        self.df_variate = self.__build_df()
        self.df_variate = self.df_variate.assign(variation=None) # Création de␣
↪la colonne variation à None
        self.n_sample_variate = n_sample
        self.n_year_variate = t_max
```

```
        u_base = self.u
        sigma_base = self.sigma
        xi_base = self.xi
        lambda_base = self.lambda_lowercase
        Lambda_base = self.lambda_uppercase
        old_n_sample = self.n_sample
        old_n_year = self.n_year

        for u in u_list :
            self.u = u
            self.sample_trajectories(n_sample=n_sample, t_max=t_max,␣
↪variate="u")
        self.u = u_base

        for sigma in sigma_list :
            self.sigma = sigma
            self.sample_trajectories(n_sample=n_sample, t_max=t_max,␣
↪variate="sigma")
        self.sigma = sigma_base

        for xi in xi_list :
            self.xi = xi
            self.sample_trajectories(n_sample=n_sample, t_max=t_max,␣
↪variate="xi")
        self.xi = xi_base

        for lambda_lowercase in lambda_list :
            self.lambda_lowercase = lambda_lowercase * np.ones(t_max)
            self.sample_trajectories(n_sample=n_sample, t_max=t_max,␣
↪variate="lambda")
        self.lambda_lowercase = lambda_base

        for lambda_uppercase in Lambda_list :
            self.lambda_uppercase = np.ones(t_max) / lambda_uppercase
            self.sample_trajectories(n_sample=n_sample, t_max=t_max,␣
↪variate="Lambda")
        self.lambda_uppercase = Lambda_base

        self.n_sample = old_n_sample
        self.n_year = old_n_year


    def plot_sample_trajectories(self, quantiles:List[float], colors:
↪List[float], R_max:float) :
        """
```

```
        Plot the damages (firt row) and the reserve (second row) for each value␣
↪of B (one column per value) \\
        The lines are the quantiles in quantiles of the average of R \\
        The colors of each line correspond to colors \\
        Also print the mean reserve for each quantile \\
        Input :
            quantiles : List[float] : list of quantile to be plotted
            colors : List[float] : list of the color of each line plotted
            R_max : float : maximum reserve value (to set the plot ylim)
        """
        if self.df.empty :
            raise ValueError("You have to simulate trajectories before plotting␣
↪them :)")
        plt.figure(figsize=(8, 8))
        for i, current_B in enumerate(self.B) :
            if i > 0 :
                print("----------------------------------------------")
            df_crop = self.df[self.df["B"] == current_B].copy()
            # Sort of the rows to find the quantiles
            df_crop = df_crop.sort_values(by=["R_mean", "tau"],␣
↪ignore_index=True)
            ruined = df_crop[df_crop["tau"] < np.inf]
            if ruined.empty :
                index_ruined = df_crop.index[0]
            else :
                index_ruined = ruined.index[0]

            for quantile, color in zip(quantiles, colors) :
                if color == "red" and quantile == 0 :
                    index = index_ruined
                    lw = 3
                else :
                    index = df_crop.index[int(self.n_sample*quantile)]
                    lw = 1
                current_S = df_crop.loc[index, "S"]
                current_R = df_crop.loc[index, "R"]
                current_R_mean = df_crop.loc[index, "R_mean"]
                print(f"B = {current_B}, mean reserve of {int(quantile*100)}th␣
↪quantile = {round(current_R_mean, 1)}")
                # Damages
                plt.subplot(2, self.nb_B, 1+i)
                plt.plot(current_S, c=color, lw=lw)
                plt.xlabel("Time (Yr)")
                plt.ylabel("Damages [S(t)]")
                # Reserve
                plt.subplot(2, self.nb_B, self.nb_B+i+1)
                plt.plot(current_R, c=color, lw=lw)
```

```
                plt.plot(current_R_mean*np.ones(self.n_year),␣
↪linestyle="dotted", c=color)
                plt.ylim((0, R_max))
                plt.xlabel("Time (Yr)")
                plt.ylabel("Reserve [R(t)]")
        plt.tight_layout()
        plt.show()

    def plot_sample_variate_params(self) :
        """
        Show the boxplot of the parameter varation
        """
        if self.df_variate.empty :
            raise ValueError("You have to simulate variations before plotting␣
↪them :)")
        for current_B in self.B :
            fig = make_subplots(rows=3, cols=2, vertical_spacing=0.1)
            df_crop = self.df_variate[self.df_variate["B"] == current_B].copy()
            df_crop["tau"] = df_crop["tau"].apply(lambda tau : min(tau, self.
↪n_year_variate+3))

            sub_df_Lambda = df_crop[df_crop["variation"] == "Lambda"]
            trace0 = go.Box(
                y=sub_df_Lambda["tau"],
                x=sub_df_Lambda["Lambda"]
            )

            sub_df_lambda = df_crop[df_crop["variation"] == "lambda"]
            trace1 = go.Box(
                y=sub_df_lambda["tau"],
                x=sub_df_lambda["lambda"]
            )

            sub_df_sigma = df_crop[df_crop["variation"] == "sigma"]
            trace2 = go.Box(
                y=sub_df_sigma["tau"],
                x=sub_df_sigma["sigma"]
            )

            sub_df_xi = df_crop[df_crop["variation"] == "xi"]
            trace3 = go.Box(
                y=sub_df_xi["tau"],
                x=sub_df_xi["xi"]
            )

            sub_df_u = df_crop[df_crop["variation"] == "u"]
            trace4 = go.Box(
```

```
                y=sub_df_u["tau"],
                x=sub_df_u["u"],
            )

            fig.append_trace(trace0, 1, 1)
            fig.append_trace(trace1, 1, 2)
            fig.append_trace(trace2, 2, 1)
            fig.append_trace(trace3, 2, 2)
            fig.append_trace(trace4, 3, 1)

            fig.update_layout(width=1000, height=800, showlegend=False,␣
↪margin=dict(t=50, b=20, r=20), title_text=f"<b>Impact of parameters for␣
↪B={current_B}</b>", title_x=0.5)
            fig.update_yaxes(range=[0, self.n_year_variate+2], title_text="Ruin␣
↪year")
            fig.update_xaxes(title_text="Return period of HW (Lambda)",␣
↪tickvals=sub_df_Lambda["Lambda"], row=1, col=1)
            fig.update_xaxes(title_text="Nb of dry days (lambda)",␣
↪tickvals=sub_df_lambda["lambda"], row=1, col=2)
            fig.update_xaxes(title_text="GPD scale",␣
↪tickvals=sub_df_sigma["sigma"], row=2, col=1)
            fig.update_xaxes(title_text="GPD shape", tickvals=sub_df_xi["xi"],␣
↪row=2, col=2)
            fig.update_xaxes(title_text="Threshold u", tickvals=sub_df_u["u"],␣
↪row=3, col=1)

            fig.show()

    def compute_proba_ruin(self) :
        """
        Print the ruin probability for each value of B
        """
        B_list = self.df["B"].unique()
        n_sample = len(self.df) / len(B_list)
        for current_B in B_list :
            df_crop = self.df[self.df["B"] == current_B]
            nb_ruined = df_crop[df_crop["tau"] < np.inf]["tau"].count() # Count␣
↪the number of tau < np.inf i.e. number of simulation ruined
            print(f"Ruin proba for B = {current_B:.4g} : {100*nb_ruined/
↪n_sample:.4g}%")

    def compute_mean_reserve_at_final_time(self) :
        """
        Print the mean reserve at t_max for each value of B
        """
        B_list = self.df["B"].unique()
```

```
        for current_B in B_list :
            df_crop = self.df[self.df["B"] == current_B]
            last_R = df_crop["R"].apply(lambda l : l[-1])
            print(f"Mean reserve at t = {self.n_year} for B = {current_B:.4g} :␣
↪{last_R.mean():.4g}")
```

3 Part 1: Article Model

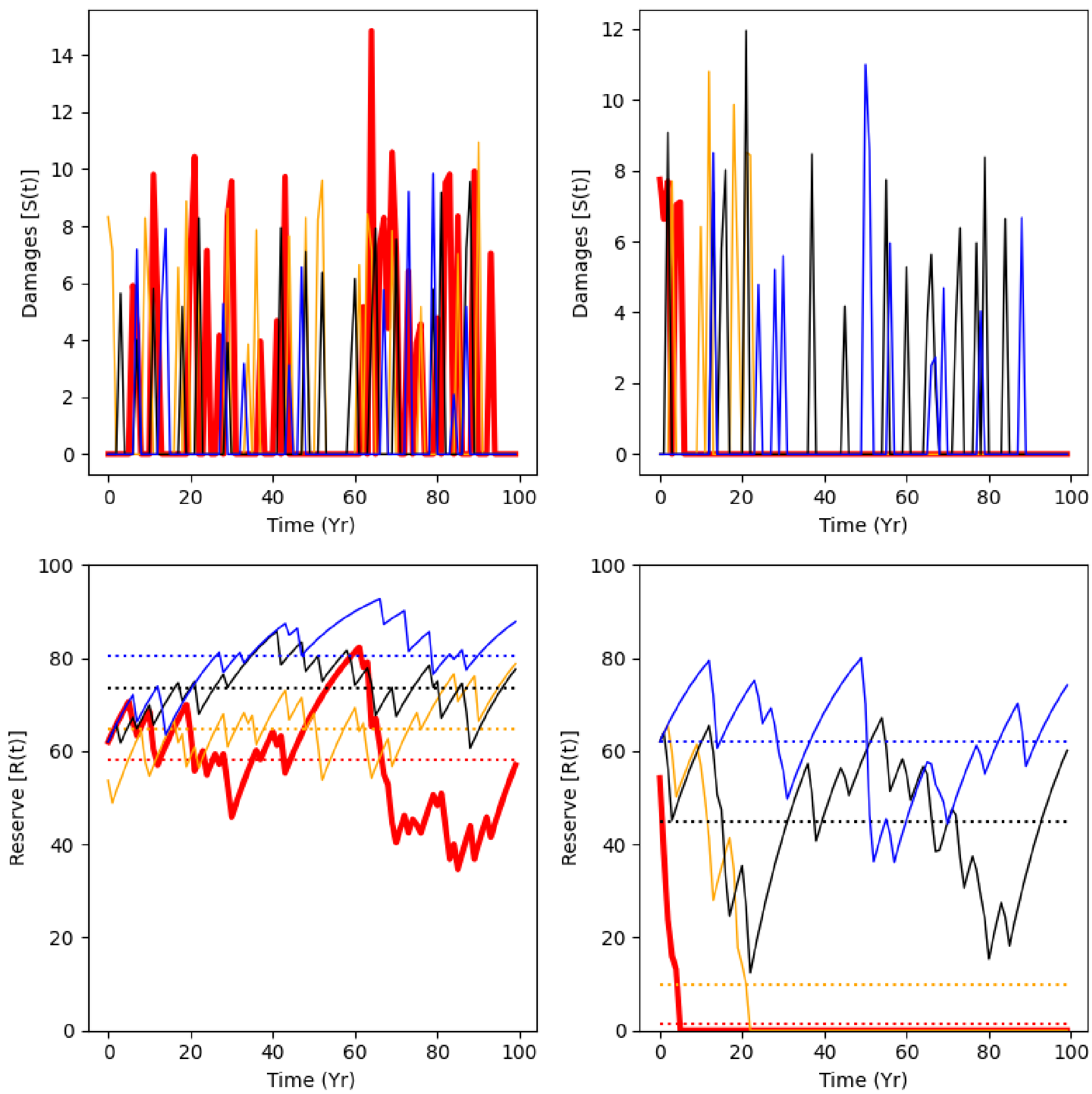
[ ]:
```
lambda_lowercase = 10 * np.ones(100)
lambda_uppercase = np.ones(100) / 5
```

[ ]:
```
params_article = dict(p0 = 5,
                      b = 0.05,
                      R_max = 100,
                      sigma = 0.1,
                      xi = -0.2,
                      u = 1,
                      compute_Ah = compute_Ah_constant,
                      lambda_lowercase = lambda_lowercase,
                      lambda_uppercase = lambda_uppercase,
                      B = [0, 1.5],
                      r0 = 60,
                      s0 = 0)

n_sample = int(1e4)
n_year = 100
trajectories_article = SimuleTrajectories(**params_article)
trajectories_article.sample_trajectories(n_sample, n_year)

quantiles = [0, 0.05, 0.5, 0.95]
colors = ["red", "orange", "black", "blue"]
trajectories_article.plot_sample_trajectories(quantiles, colors,␣
↪params_article["R_max"])
```

```
B = 0, mean reserve of 0th quantile = 58.1
B = 0, mean reserve of 5th quantile = 64.8
B = 0, mean reserve of 50th quantile = 73.6
B = 0, mean reserve of 95th quantile = 80.6
---------------------------------------------
B = 1.5, mean reserve of 0th quantile = 1.5
B = 1.5, mean reserve of 5th quantile = 9.9
B = 1.5, mean reserve of 50th quantile = 44.9
B = 1.5, mean reserve of 95th quantile = 62.2
```



[ ]:
```
trajectories_article.compute_mean_reserve_at_final_time()
trajectories_article.compute_proba_ruin()
```

```
Mean reserve at t = 100 for B = 0 : 77.73
Mean reserve at t = 100 for B = 1.5 : 32.42
Ruin proba for B = 0 : 0%
Ruin proba for B = 1.5 : 33.71%
```

4 Part 2: Change Ah to be function of B

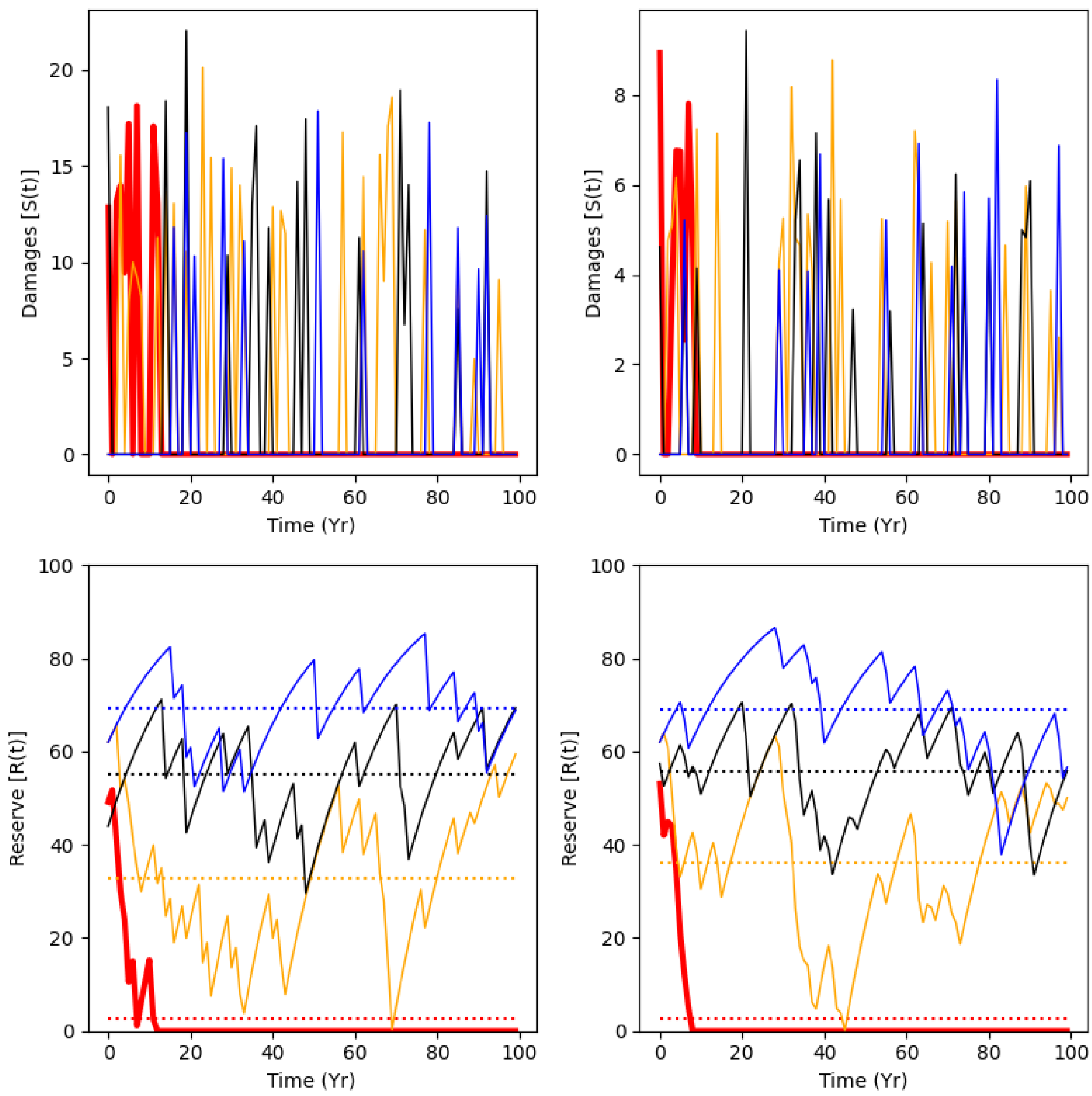
[ ]:
```
lambda_lowercase = 10 * np.ones(100)
lambda_uppercase = np.ones(100) / 5
```

[ ]:
```
params_Ah_function = dict(p0 = 5,
                          b = 0.05,
                          R_max = 100,
                          sigma = 0.1,
                          xi = -0.2,
                          u = 1,
                          compute_Ah = compute_Ah,
                          lambda_lowercase = lambda_lowercase,
                          lambda_uppercase = lambda_uppercase,
                          B = [0, 1.5],
                          r0 = 60,
                          s0 = 0)


n_sample = int(1e4)
n_year = 100
trajectories_Ah_function = SimuleTrajectories(**params_Ah_function)
trajectories_Ah_function.sample_trajectories(n_sample, n_year)

quantiles = [0, 0.05, 0.5, 0.95]
colors = ["red", "orange", "black", "blue"]
trajectories_Ah_function.plot_sample_trajectories(quantiles, colors,␣
  ↪params_Ah_function["R_max"])
```

```
B = 0, mean reserve of 0th quantile = 2.6
B = 0, mean reserve of 5th quantile = 32.7
B = 0, mean reserve of 50th quantile = 55.1
B = 0, mean reserve of 95th quantile = 69.3
---------------------------------------------
B = 1.5, mean reserve of 0th quantile = 2.6
B = 1.5, mean reserve of 5th quantile = 36.0
B = 1.5, mean reserve of 50th quantile = 55.6
B = 1.5, mean reserve of 95th quantile = 69.1
```



[ ]:
```
trajectories_Ah_function.compute_mean_reserve_at_final_time()
trajectories_Ah_function.compute_proba_ruin()
```

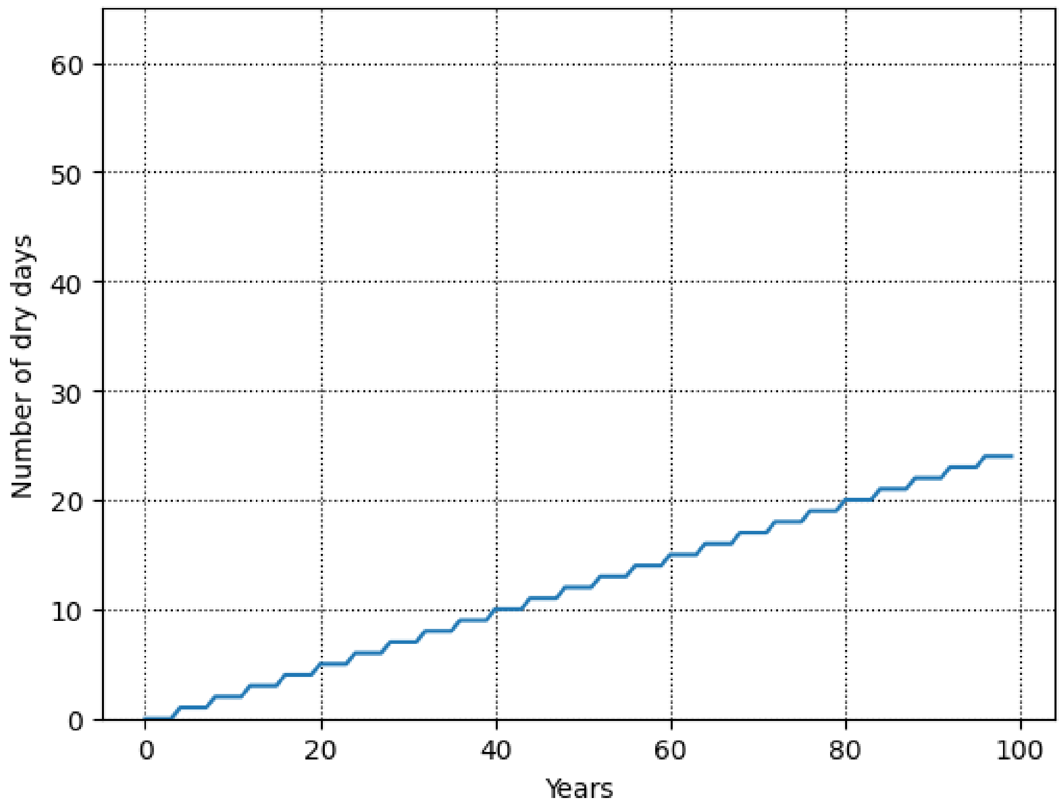
```
Mean reserve at t = 100 for B = 0 : 53.58
Mean reserve at t = 100 for B = 1.5 : 52.49
Ruin proba for B = 0 : 6.32%
Ruin proba for B = 1.5 : 5.32%
```

5 Part 3: Taking climate change into account

5.1 A. Increasing number of dry days

[ ]:
```
lambda_lowercase = np.array([0 + 2.5 * i//10 for i in range(100)])
lambda_uppercase = np.ones(100) / 5
```

[ ]:
```
plt.plot(lambda_lowercase)
plt.ylabel("Number of dry days")
plt.xlabel("Years")
plt.ylim([0, 65])
plt.grid(linestyle="dotted", c="black")
```



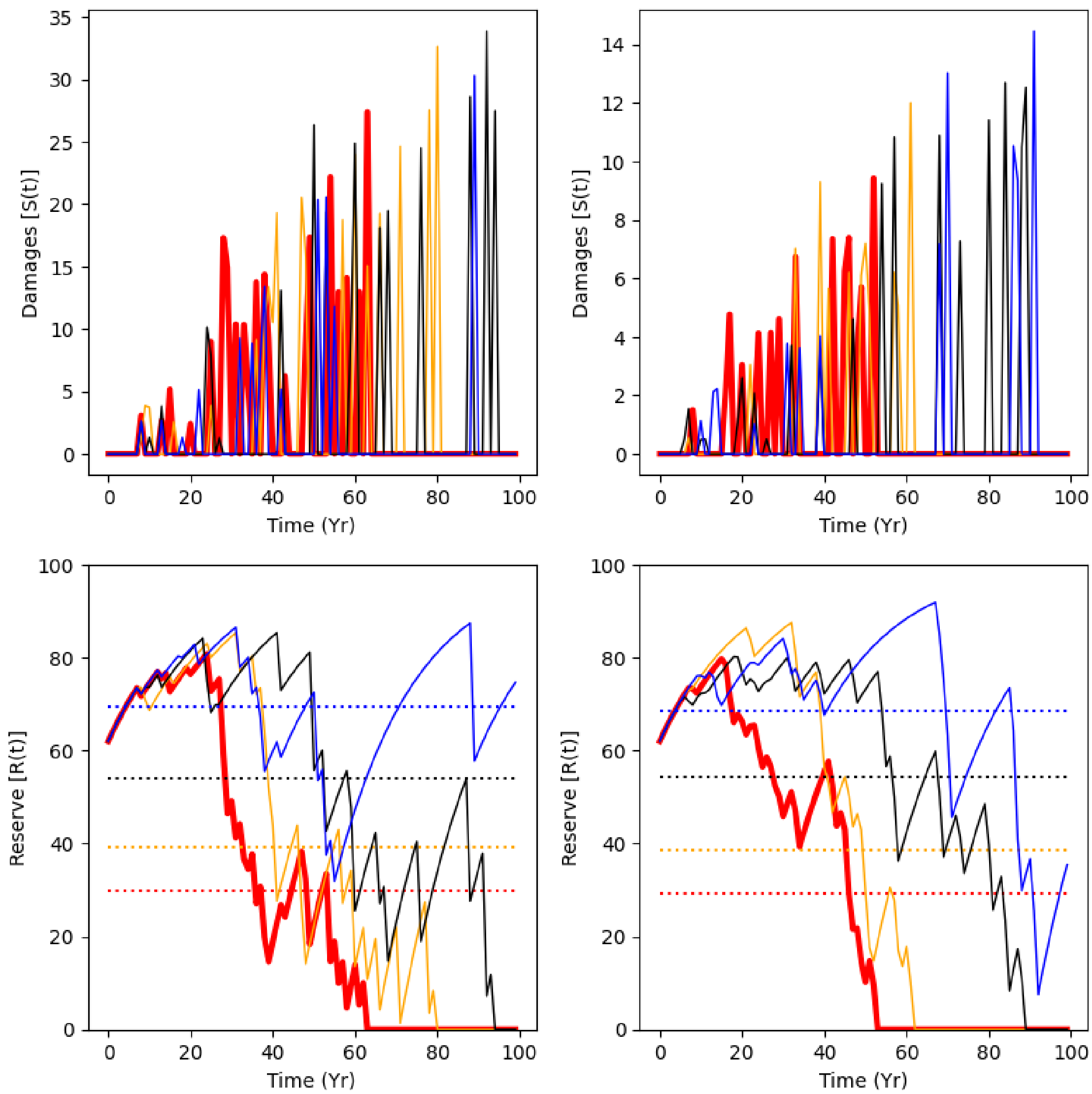
[ ]:
```
params_ipp_lambda_lowercase = dict(p0 = 5,
                                   b = 0.05,
                                   R_max = 100,
                                   sigma = 0.1,
                                   xi = -0.2,
                                   u = 1,
                                   compute_Ah = compute_Ah,
                                   lambda_lowercase = lambda_lowercase,
```

```
                                   lambda_uppercase = lambda_uppercase,
                                   B = [0, 1.5],
                                   r0 = 60,
                                   s0 = 0)

n_sample = int(1e4)
n_year = 100
trajectories_ipp_lambda_lowercase =␣
 ↪SimuleTrajectories(**params_ipp_lambda_lowercase)
trajectories_ipp_lambda_lowercase.sample_trajectories(n_sample, n_year)

quantiles = [0, 0.05, 0.5, 0.95]
colors = ["red", "orange", "black", "blue"]
trajectories_ipp_lambda_lowercase.plot_sample_trajectories(quantiles, colors,␣
 ↪params_Ah_function["R_max"])
```

```
B = 0, mean reserve of 0th quantile = 29.9
B = 0, mean reserve of 5th quantile = 39.2
B = 0, mean reserve of 50th quantile = 54.1
B = 0, mean reserve of 95th quantile = 69.4
---------------------------------------------
B = 1.5, mean reserve of 0th quantile = 29.2
B = 1.5, mean reserve of 5th quantile = 38.6
B = 1.5, mean reserve of 50th quantile = 54.5
B = 1.5, mean reserve of 95th quantile = 68.5
```



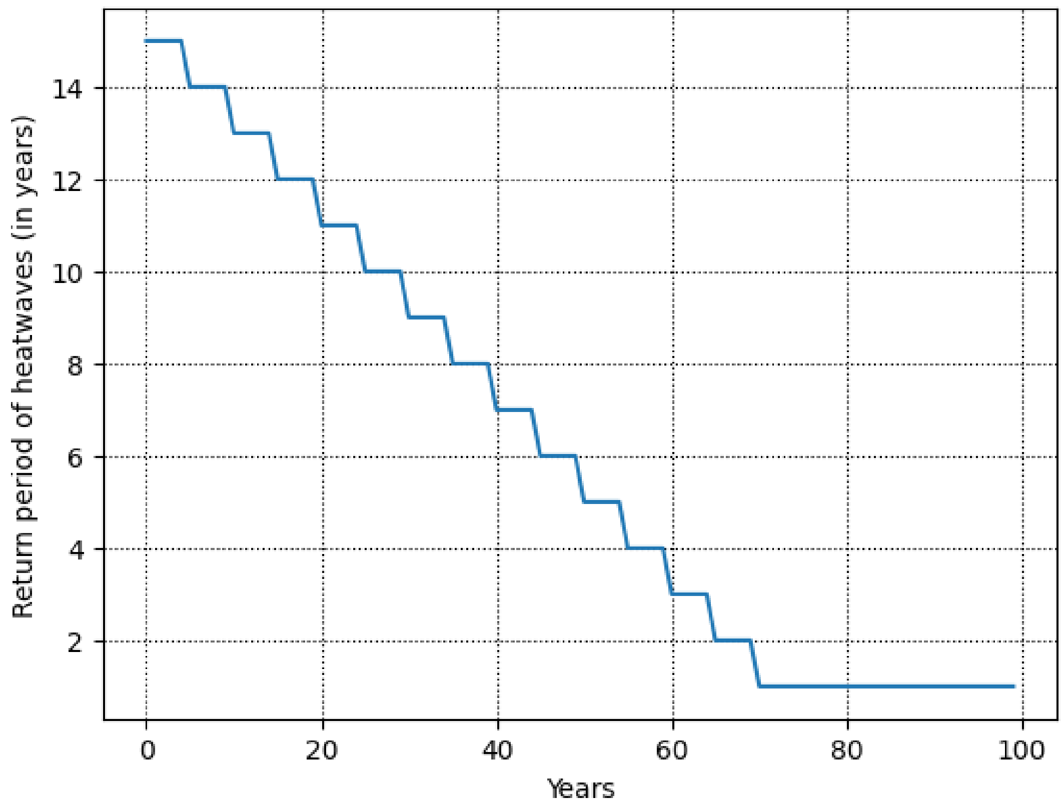

```
[ ]: trajectories_ipp_lambda_lowercase.compute_mean_reserve_at_final_time()
     trajectories_ipp_lambda_lowercase.compute_proba_ruin()
```

```
Mean reserve at t = 100 for B = 0 : 14.62
Mean reserve at t = 100 for B = 1.5 : 14.6
Ruin proba for B = 0 : 69.2%
Ruin proba for B = 1.5 : 65.36%
```

5.2 B. Decreasing return period of heatwaves

[ ]:
```
lambda_lowercase = np.array([0 + 2.5 * i//10 for i in range(100)])
lambda_uppercase = 1 / np.array([max(15 - 2 * i//10, 1) for i in range(100)])
```

[ ]:
```
plt.plot(1/lambda_uppercase)
plt.ylabel("Return period of heatwaves (in years)")
plt.xlabel("Years")
plt.grid(linestyle="dotted", c="black")
```



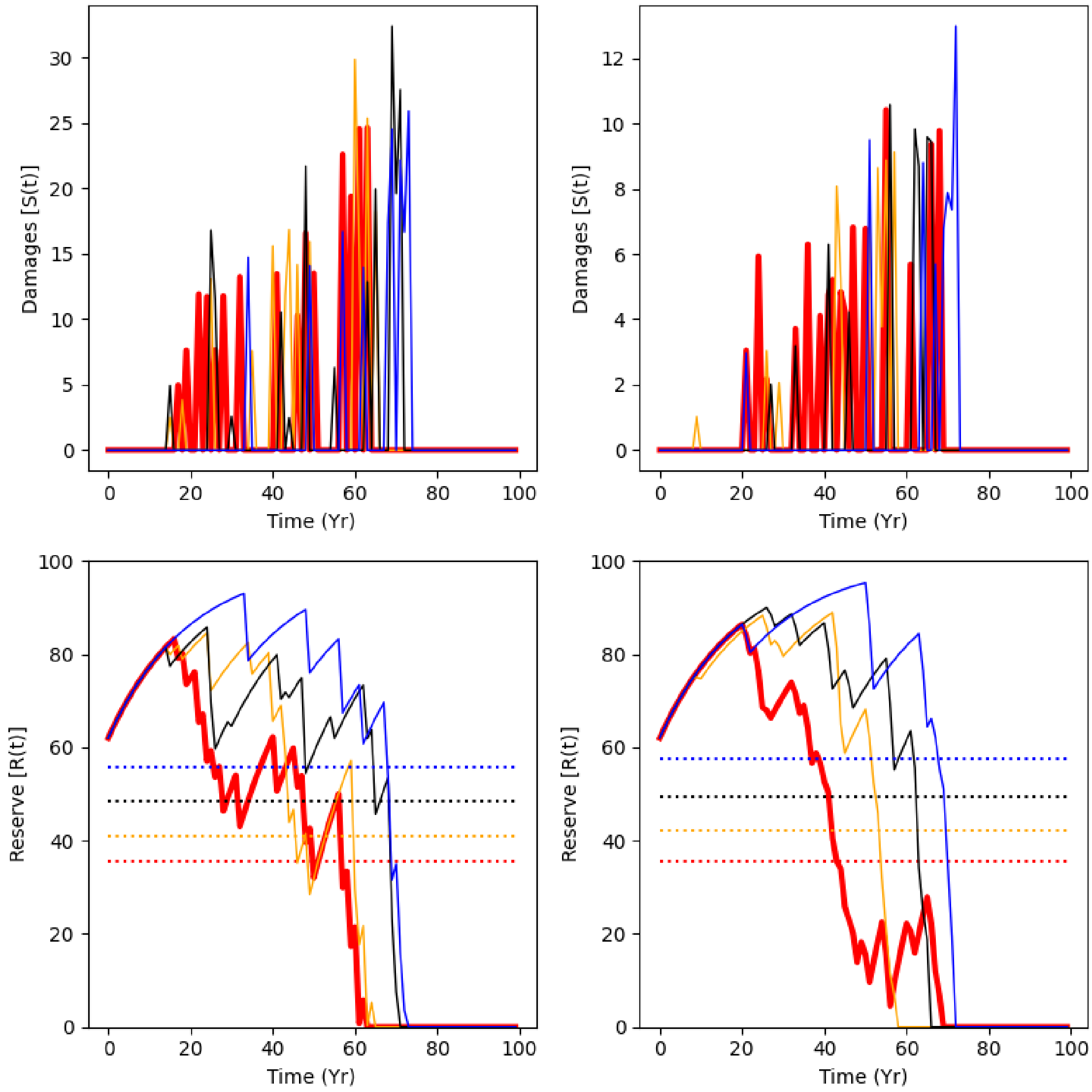
[ ]:
```
params_ipp_both_lambda = dict(p0 = 5,
                              b = 0.05,
                              R_max = 100,
                              sigma = 0.1,
                              xi = -0.2,
                              u = 1,
                              compute_Ah = compute_Ah,
                              lambda_lowercase = lambda_lowercase,
                              lambda_uppercase = lambda_uppercase,
                              B = [0, 1.5],
                              r0 = 60,
```

```
                              s0 = 0)

n_sample = int(1e4)
n_year = 100
trajectories_ipp_both_lambda = SimuleTrajectories(**params_ipp_both_lambda)
trajectories_ipp_both_lambda.sample_trajectories(n_sample, n_year)

quantiles = [0, 0.05, 0.5, 0.95]
colors = ["red", "orange", "black", "blue"]
trajectories_ipp_both_lambda.plot_sample_trajectories(quantiles, colors,
 ↪params_Ah_function["R_max"])
```

```
B = 0, mean reserve of 0th quantile = 35.4
B = 0, mean reserve of 5th quantile = 41.0
B = 0, mean reserve of 50th quantile = 48.6
B = 0, mean reserve of 95th quantile = 55.7
---------------------------------------------
B = 1.5, mean reserve of 0th quantile = 35.5
B = 1.5, mean reserve of 5th quantile = 42.0
B = 1.5, mean reserve of 50th quantile = 49.5
B = 1.5, mean reserve of 95th quantile = 57.4
```



[ ]:
```
trajectories_ipp_both_lambda.compute_mean_reserve_at_final_time()
trajectories_ipp_both_lambda.compute_proba_ruin()
```

```
Mean reserve at t = 100 for B = 0 : 0
Mean reserve at t = 100 for B = 1.5 : 0
Ruin proba for B = 0 : 100%
Ruin proba for B = 1.5 : 100%
```

6 Part 4: Influence of each parameter on the ruin time

Takes about 27min to run as it cannot be parallelized

[ ]:
```
trajectories_variation = SimuleTrajectories(**params_Ah_function)   # hPP␣
↪parameters with Ah function (most representative case without iPP)
                                                                     # Beacuse␣
↪sample_variate_params only take hPP values for lambda ans Lambda
params_variation = dict(u_list = np.arange(1, 6),
                        sigma_list = np.arange(0.08, 0.21, 0.02),
                        xi_list = np.arange(-0.45, -0.19, 0.05),
                        lambda_list = np.arange(2, 31),
                        Lambda_list = np.arange(2, 16))


trajectories_variation.sample_variate_params(int(1e4), 100, **params_variation)
```

[ ]:
```
trajectories_variation.plot_sample_variate_params()
```