```
# ------------------------------------------------------------------------------------- #
#                                  spont_decay_v2.py                                    #
#                          -- radioactive spontaneous decay --                          #
#                             applying a monte-carlo method                             #
#                                                                                       #
#                             by Kostas Tsachalinas, 2022                               #
#                                                                                       #
#  - main idea based on listing #6.6 (p.282) of 'Computational Problems for Physics' book #
#                                                                                       #
#       but heavily modified and enriched, by using python's vectorizing techniques,    #
#        making some calculations, as well as plotting output data, regression line     #
#                           and generated pseudo-randoms histogram                      #
#                                                                                       #
# ------------------------------------------------------------------------------------- #

import numpy as np
import matplotlib.pyplot as plt
import random

seed = 200                              # set the seed of python's pseudo-randoms generator
random.seed(seed)

lambda1 = 0.005                         # probabilistic spontaneous decay rate parameter λ:
                                        # probability of a single atom decaying @ a certain time-step

N = 100000                              # initial number of radioactive atoms population

time_max = 3000                         # max number of mc-time steps for current simulation,
                                        # -> max possible length of discrete-time vector

tV = np.linspace(0,time_max, time_max)  # create time-vector of successive points (start, end & number of points)
tV = tV.astype(int)                     # convert to vector of integers, for discrete numbering m.c. steps in time-evolution loop

rem_atomsV = np.zeros(time_max)         # create vector of remaining atoms @ each t-step, of length time_max (init all vals to 0)

plot_histogram = 1                      # flag to/not-to plot histogram of [0,1) pseudo-randoms distribution

if (plot_histogram):                    # this will take some time, depending on atoms number N !
  rndV = np.zeros(int(N*time_max/10))   # create & init vector for storing pseudo-randoms, of adequate size

mc_tries = 0                            # monte-carlo tries number, init val

remaining_atoms = N                     # current number of non-decayed atoms, fed as upper limit of (decaying) atoms sweeping loop
                                        # @ a given time-step (default: N, for time=0)

rem_atomsV[0] = N                       # init: remaining atoms @ simulation start (t=0), stored @ vector

t_end = time_max                        # time vector upper limit, to be used for calculations & plotting (default: time_max)
zo_time = time_max                      # init val of population zero-out time, exact val to be calculated later on..
delta_from_half = 50.0                  # init val of current population distance from 50%, for half-life time estimation

print(" time = 0  | non-decayed population :", N,"/",N,"atoms ( 100 % ) ") # print population initial condition

# ------------------------------------------------------------------------------------------------------------------------
for time in tV[1:]:                     # time evolution loop: current time taken out of mc-steps time vector tV (tV[0]=0)

    # At this mc time-step, some atoms of the remaining population may stochastically decay:
    for atom in range(1, remaining_atoms + 1):

        my_random = random.random()                     # get a pseudo-random float in [0,1) from python's pseudorandoms generator

        if (my_random < lambda1):                       # probabilistic rule for stochastic radioactive decay of a certain atom
            remaining_atoms -= 1                        # -> revise non-decayed atoms population respectively

        if (plot_histogram):
            rndV[mc_tries] = my_random                  # store this pseudo-random to vector, to do some graph later on
        mc_tries += 1                                   # increase index of mc tries
    # --------------------------------

    rem_atomsV[time] = remaining_atoms                  # store current population number @ vector indexed by mc time-steps

    #-------------------------------------------------
    # Estimating half-life time, directly from simulation:
    tmp_delta_from_half = delta_from_half               # store previous val, for comparing with current val

    rem_perc = float(remaining_atoms)*100.0/float(N)    # percentage (%) of remaining over initial population

    delta_from_half = np.abs(rem_perc - 50.0)           # percentage distance from 50%

    if (delta_from_half < tmp_delta_from_half):         # get an even better aproach to half-life time value
      half_life_time = time
    # -------------------------------------------------
    formatted_perc ='{:.3}'.format(rem_perc)            # string format: set population percentage @ 3 decimal places
    mc_tries_sci_not = '{:.4e}'.format(mc_tries)        # string format: set monte-carlo tries number @ sci notation (4 decimal
places)

    print(" time =",time," | non-decayed population :",remaining_atoms,"/",N, "atoms (", formatted_perc ,"% )  m.c.
tries :",mc_tries_sci_not)

    if (remaining_atoms == 0):
      zo_time = time                                    # population zero-out time
      break

print("\n total monte-carlo tries involved :", mc_tries)#all mc tries:(successful/unsuccessful)
print("\n half-life time = ",half_life_time, "mc time-steps (directly from simulation)")


# LINEAR REGRESSION:
# ------------------------------------------------------------------------------------
if (zo_time < time_max):
    t_end = zo_time                             # to be used @ plotting time subvector

regr_last = 1500                                # upper limit index of acceptable x-axis data points, for linear regression

if (regr_last > zo_time):                       # just as not to assume more input data than actual ones!
  regr_last = zo_time - 1
```

```
log_rem_atomsV = np.log(rem_atomsV[:regr_last])     # transform y-axis vals to log scale

# Fit chosen data by linear regression polynomial:
# slope, intercept = np.polyfit(xV, yV, n)          # here, degree of polynomial n = 1 (least-squares line)
a, b = np.polyfit(tV[:regr_last], log_rem_atomsV, 1)

print("\n regression line slope :", a,"\n actual decay-rate constant :", lambda1,"(time-step^-1)\n rel. error : ",
np.abs( (np.abs(a)-lambda1)*100.0/lambda1),"%\n" )


# PLOTS:
# ------------------------------------------------------------------------------
# this figure, (#1), will overlay both mc-simulation data & respective regression line:
fig1 = plt.figure()

plt.yscale('log')                                 # setting log scale @ y-axis (semi-log diagram)

yyV = np.exp(a * tV[:t_end-1] + b)                # create regression line points vector and exponentiate,
                                                  # so as regression line will emerge @ semi-log diagram

# ------------------------------------------------------------------------------------------
# Regression line points scatterplot (plot up to t_end-1, to avoid plotting log[n=0] @ t_end):
plt.scatter(tV[:t_end-1], yyV, color='gray', marker='.',label='L-S linear regression line')

# Remaining atoms data vs. mc-time scatterplot:
plt.scatter(tV[:t_end-1], rem_atomsV[:t_end-1], color='red', marker='.',label='Monte-Carlo simulation')

plt.grid(color='gray', linestyle='--', linewidth=2)
plt.title('Radioactive spontaneous decay monte-carlo simulation\n(regression line slope: %f)'%a)
plt.xlabel('Time (MC steps)')
plt.ylabel('Non-decayed atoms population, N(t)')
plt.legend()

# ------------------------------------------------------------------------------------------------------------------------
# this figure, (#2), will plot a frequencies histogram of all randoms used, to verify their unifom distribution in [0,1):
if (plot_histogram):
  rnd_effV = rndV[:mc_tries]    # use mc-tries number indexing subvector up to population zero-out (effective randoms vector)

  fig2 = plt.figure()

  #n, bins, patches = plt.hist(xV, bins='auto')
  plt.hist(rnd_effV, bins='auto')

  plt.grid(axis='y')  # set grid on y-axis
  plt.xlabel('pseudo-random float in [0,1)')
  plt.ylabel('counts')
  plt.title('%s uniformly-distributed pseudo-randoms freq. histogram,\nused @ this monte-carlo simulation'%mc_tries_sci_not)

  # ------------------------------------------------------------------------------------------------------
  # this (optional) figure, (#3), will plot a scatterplot of all [0,1) pseudo-randoms used vs. their indices
  # Notice: Big data -> very slow! Use only @ small initial atoms population N!

  plot_rnd_scattterpl = 0
  if (plot_rnd_scattterpl):

    fig3 = plt.figure()

    rnd_indexV= np.arange(1,len(rnd_effV)+1)

    plt.scatter(rnd_indexV, rnd_effV,marker='.')

    plt.grid(axis='x')  # set grid on x-axis
    plt.grid(axis='y')  # set grid on y-axis
    plt.xlabel('pseudo-random index #')
    plt.ylabel('pseudo-random float')
    plt.title('%s uniformly-distributed pseudo-randoms scatterplot,\nused @ this monte-carlo simulation'%mc_tries_sci_not)

plt.show()   # show all created figures. User must close all figure windows, so as this program will terminate !

print("  -- Program end --\n")
```