MC_reliability_simulations

June 17, 2021

1 Monte Carlo simulations for space systems reliability

1.1 Introduction

An interesting application of the Monte Carlo method is the comparison of different system architectures and choosing the optimal one for a given mission. While comparisons can be made using analytical formulas, Monte Carlo simulations can indicate deviations from the expected behaviors and since they usually have a specified failure condition which is based on logic transitions, they enable the investigation of complex systems which have extremely complicated definitions for the reliability. In general, the reliability function $R(t)$ can be defined as the probability of a component/system not failing until the time step t. This is why reliability is also commonly defined as:

$$R(t) = 1 - F(t)$$

where $F(t)$ corresponds to the cumulative distribution function for the component/system. For electronic systems, the distribution which better interprets the data is the **exponential distribution**, resulting in:

$$F(t) = \int_0^t f(x)dx = 1 - e^{-\lambda t}$$

and finally

$$\mathrm{R(t) = 1 - F(t) = e}^{-\lambda t}$$

which enables us to calculate the reliability of an electronic component at time t, given the failure rate λ of the component.

1.2 Space environment

Space systems in Earth orbits are subject to different kinds of radiation environments. These include:

1. **Van Allen belts**
    - Trapped protons and electrons (internal belt)
    - Trapped electrons (external belt)
2. **Solar energetic particles**

- Electrons and protons/heavy ions (solar flares)
- Protons and heavy ions (Coronal Mass Ejections)

3. **Cosmic galactic rays**
   - Protons and heavy ions

These particles can have a negative effect on systems operating in earth orbits. Specifically they can cause several types of damages, including:

1. Displacement damage
2. Total Ionizing dose damage
3. **Single Events**

In this assignment we are interested in single events, which due to their random nature can be modeled very accurately using Monte Carlo simulations so that we can compare architectures for a spacecraft's On-board processing system. Specifically, we will investigate micro-controller units and determine **which architecture is the most preferable at the orbit environment of the International Space Station**. The four architectures which we are going to compare are:

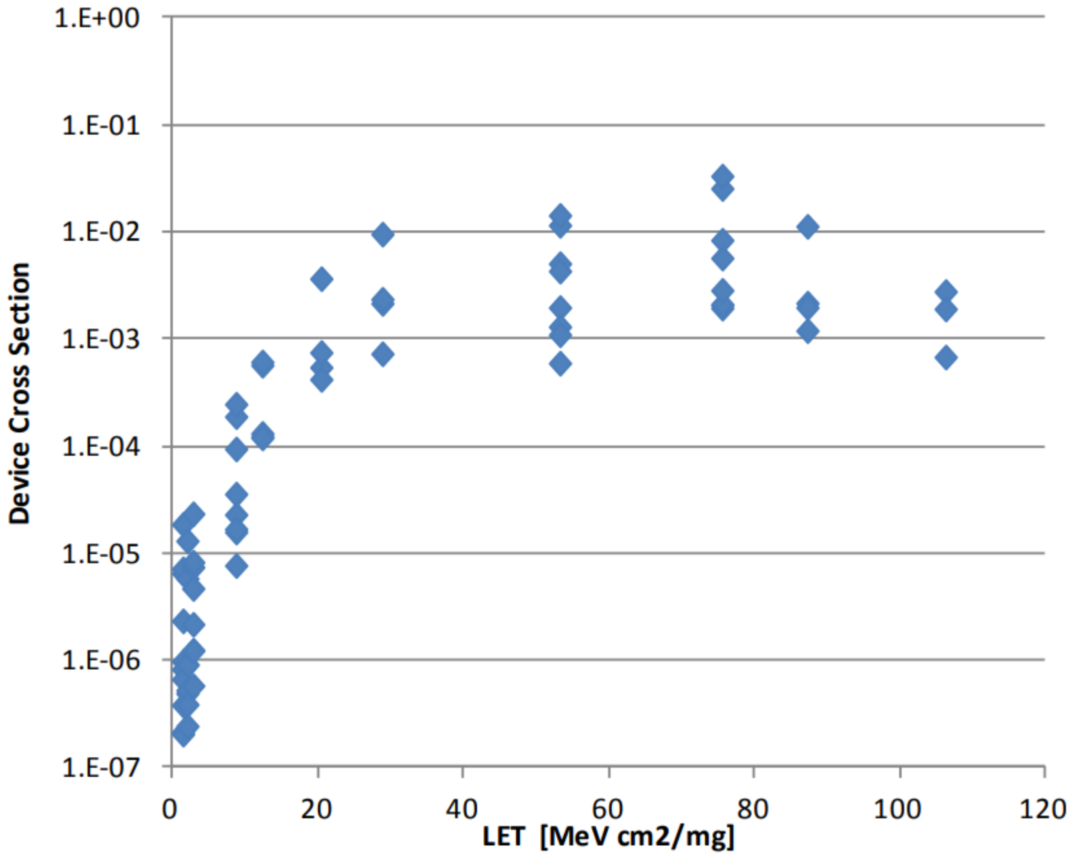
1. Single MCU
2. Dual Redundant MCU
3. Triple Modular Redundancy with immune voter
4. Triple Modular Redundancy with non-immune voter

1.3 OMERE/Failure rates due to radiation effects

OMERE is a free program by TRAD which enables users to calculate dose estimates and susceptibility of components to radiation while in orbit. For more accurate results, a user can input radiation testing data instead of built in values. We will utilize 2 main components:

1) **Micro-controller Cypress EZ-USB FX2**, which has the functionality required to perform tasks in small satellites

The test report for Cypress EZ-USB FX2 can be found here. The main result from this test report is the cross section vs LET plot.



LET [MEV cm^2/mg]	Device Cross Section
1.818182	9.55E-07
2.109091	3.67E-07
2.981818	4.5E-06
8.8	1.54E-05
8.945455	9.34E-05
12.43636	0.000579
20.72727	0.000761
20.72727	0.000393
28.58182	0.000679
28.87273	0.009775
29.30909	0.002272
52.87273	0.000579
53.30909	0.014402
53.45455	0.001071

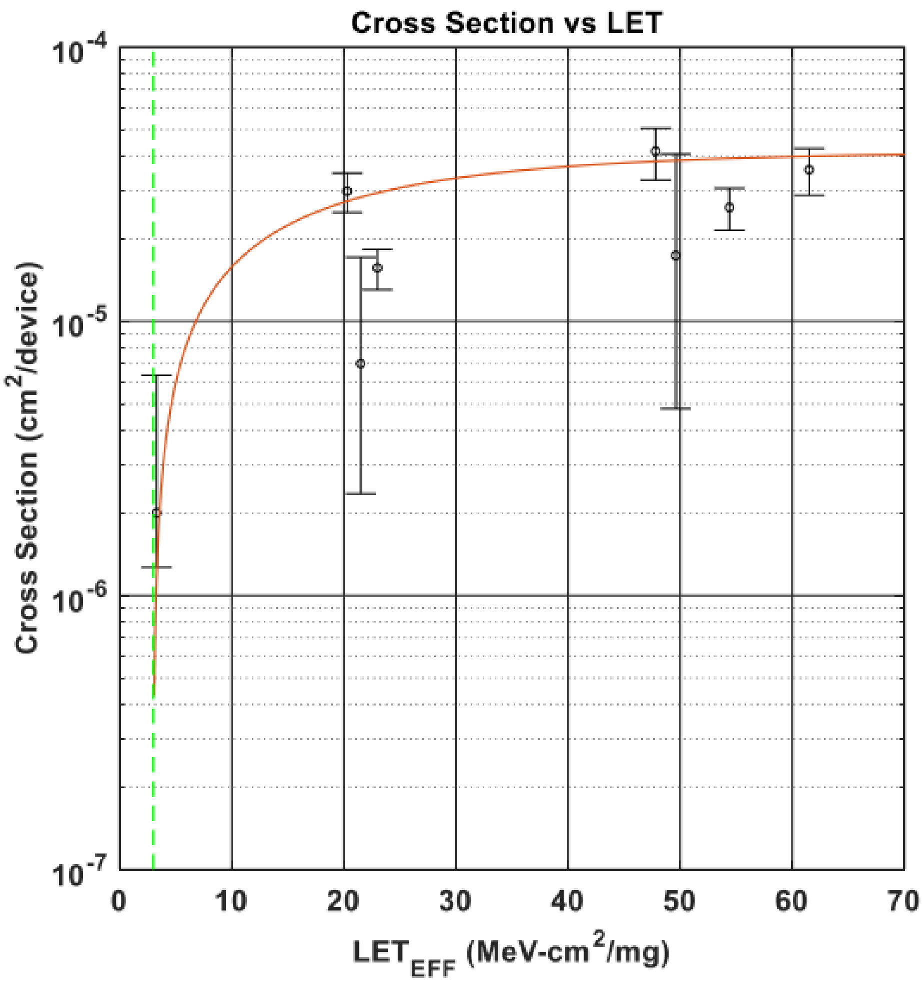
LET [MEV cm^2/mg]	Device Cross Section
75.70909	0.005403
75.70909	0.025465
75.85455	0.001851
87.49091	0.011207
87.78182	0.001937
106.2545	0.000694
106.8364	0.00292

Fitting a Weibull distribution to this data and providing it to OMERE, we can calculate a failure rate of:

$$\lambda_{MCU} = 3.17 \times 10^{-4} \, \frac{failures}{day}$$

2) **Microcontroller Texas Instruments MSP430FR5969-SP**, which offers very limited functionality and thus can be selected as a voter circuit.

The test report for MSP430FR5969-SP can be found here here. The main result from this test report is the cross section vs LET plot.



LET [MEV cm^2/mg]	Device Cross Section
3.882127	8.05E-07
20.83679	4.91E-06
22.00243	1.86E-06
23.48596	3.19E-06
48.38812	6.14E-06
50.18956	3.4E-06
54.95806	4.47E-06
62.05782	5.53E-06
68.94565	5.99E-06

In both cases these data are fitted to a Weibull distribution to be used as input to OMERE to compute a realistic response to heavy ions. OMERE will then return the expected Single Event

Effects rate per day.

Fitting a Weibull distribution to this data and providing it to OMERE, we can calculate a failure rate of:

$$\lambda_{voter} = 3.13 \times 10^{-6} \, \frac{failures}{day}$$

1.4 Random number generator

In order for us to conduct a Monte Carlo simulation, we need random numbers to determine the system's behavior. For convenience, these random numbers have to be distributed uniformly within the interval $[0, 1]$. We will implement a dedicated random number generator using the linear repeat method (or the so called Linear congruential method). This random number generator is based on the general relationship:

$$x_{n+1} = [a_0 x_n + a_1 x_{n-1} + \cdots + a_j x_{n-j} + b] \bmod M$$

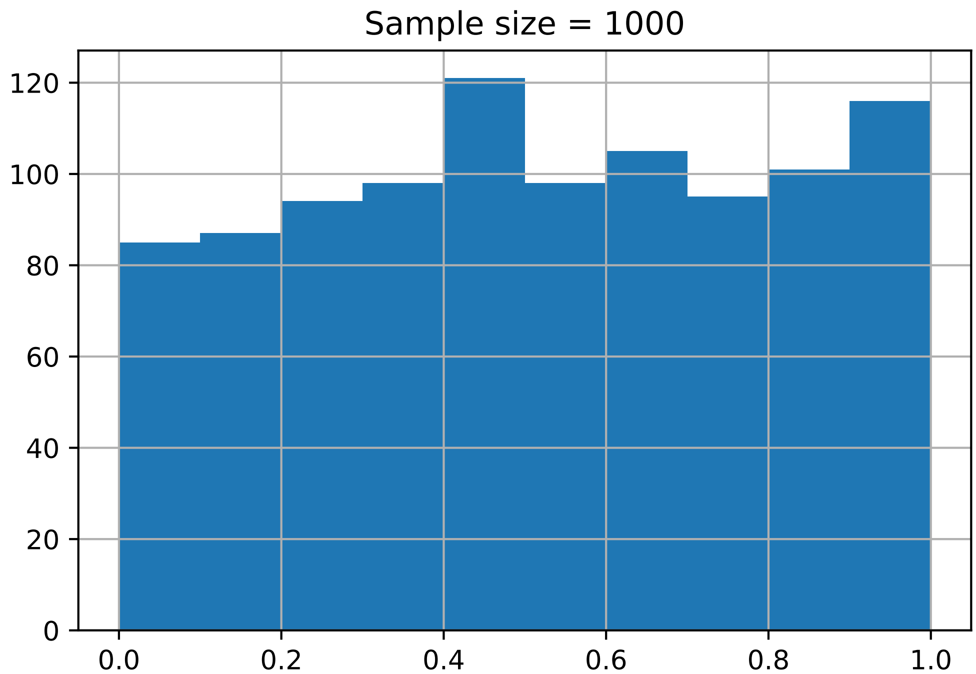
to simplify things, we will use the relationship:

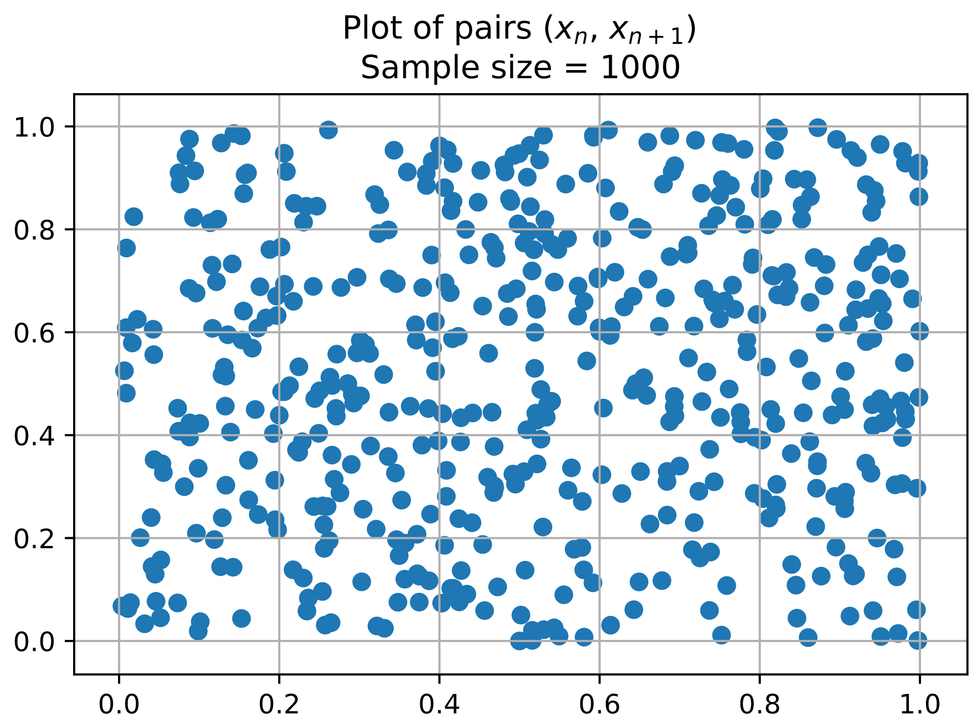
$$x_{n+1} = [a x_n + b] \bmod M$$

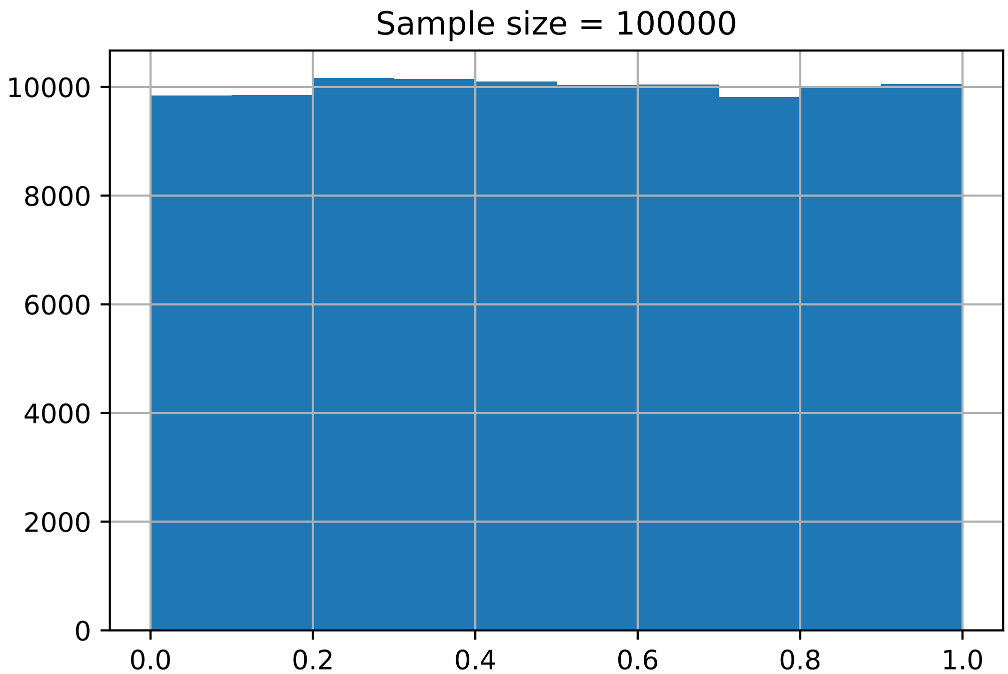
where M has to be a very large number (since we are using a 64 bit system we will use $M = 2^{64} - 1$), $a = 1812433253$ and b has to be an odd number. We will also check the behavior of our random number generator. An obvious downside is that since the generation of a new random number requires the previous one, the numbers are correlated (which is an issue for smaller values of a and M.

[1]:
```
import matplotlib.pyplot as plt
# Make jupyter export images as .pdf files for higher quality
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')


def linear_congruential_generator(xn_prev, N):
    numbers_list = []
    for i in range(0, N):
        a = 1812433253
        M = pow(2, 64) - 1
        b = 89539
        xn_next = (a*xn_prev+b) % M
        numbers_list.append(xn_next/M)
        xn_prev = xn_next
    return numbers_list


numbers = 1000
result = linear_congruential_generator(0.02, numbers)
```

```
plt.figure()
plt.hist(result)
plt.title("Sample size = %d" % numbers)
plt.grid()

x_vector = [0.5]
y_vector = [0.0]
for i in range(0, numbers-1):
    if i == 0:
        y_vector[0] = result[i]
    elif i%2==0:
        y_vector.append(result[i])
    else:
        x_vector.append(result[i])

plt.figure()
plt.scatter(x_vector, y_vector)
plt.title("Plot of pairs ($x_{n}$, $x_{n+1}$)\nSample size = %d" % numbers)
plt.grid()
plt.show()

numbers = 100000
result = linear_congruential_generator(0.02, numbers)
plt.figure()
plt.hist(result)
plt.title("Sample size = %d" % numbers)
plt.grid()
plt.show()

c = []
for i in range(1, 6):
    sum = 0
    for j in range(0, numbers-i):
        sum += result[j]*result[j+i]
    c.append(sum/(j+1))

print(c)
```


Sample size = 1000
120
100
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0


Plot of pairs (x_n, x_{n+1})
Sample size = 1000
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0




```
[0.2506476679042393, 0.25050892233902305, 0.2506513026382199,
0.2504787236977658, 0.2504478901170785]
```

There are two things to take away from this random number generator

1) By plotting (x_n, x_{n+1}) in pairs, no apparent correlation between points is apparent. Based only on this, we can conclude that our numbers are random.
2) However, if the random variables X and Y are independent and uniform in $[0, 1]$, then the mean for their product shall be:

$$\langle xy \rangle = \int_0^1 \int_0^1 xydxdy = \int_0^1 xdx \int_0^1 ydy = \frac{1}{4}$$

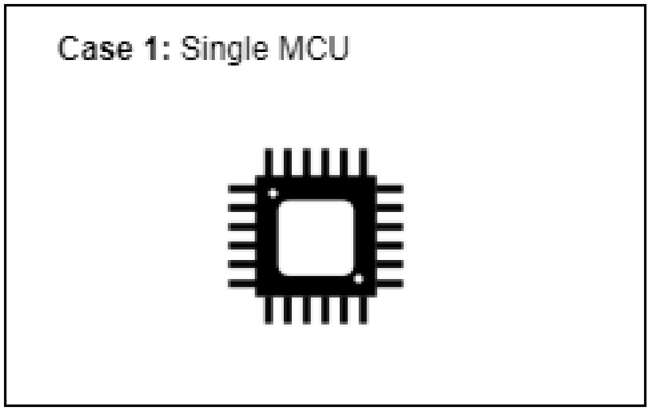
which we can check by defining a sequence of random numbers x_i and $y_i = x_{i+k}$ and calculate the average

$$c(k) = \frac{1}{N} \sum_{i=1}^{N} x_i x_{i+k}, \quad k = 1, 2, 3, \cdots$$

Since we can observe that for all cases we have $c(k) \simeq \frac{1}{4}$ the numbers seem to be uncorrelated. But, since the error is higher than $\frac{1}{\sqrt{N}}$, there is a measurable correlation between our points. This is why we will refer to our numbers as pseudo-random.

1.5 Case 1: Single MCU

The architecture is the following



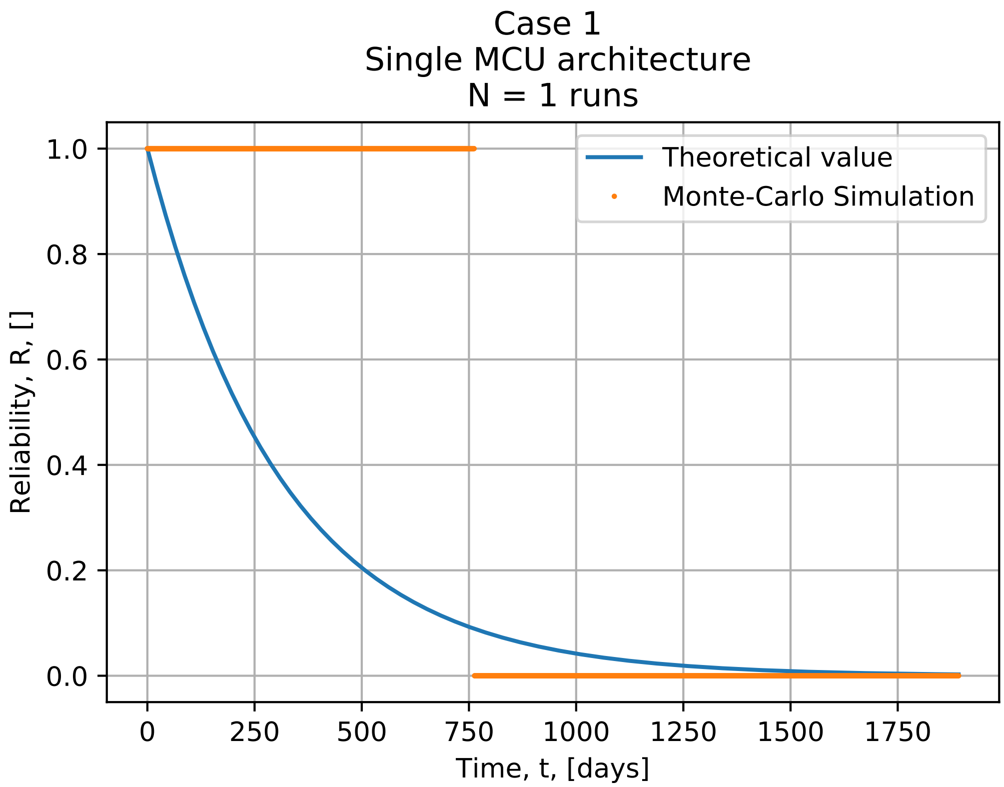
For a single MCU, the system reliability is equal to the component reliability, namely:

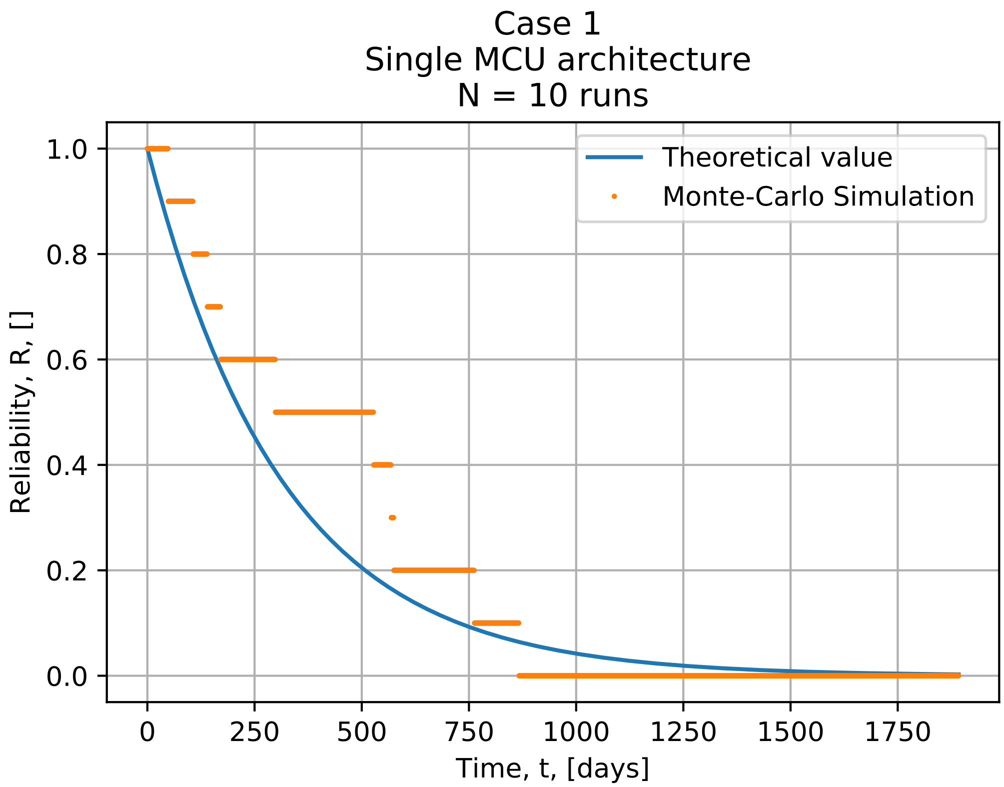
$$\mathrm{R}_{\mathrm{system}}(\mathrm{t}) = \mathrm{e}^{-\lambda_{MCU} t}$$

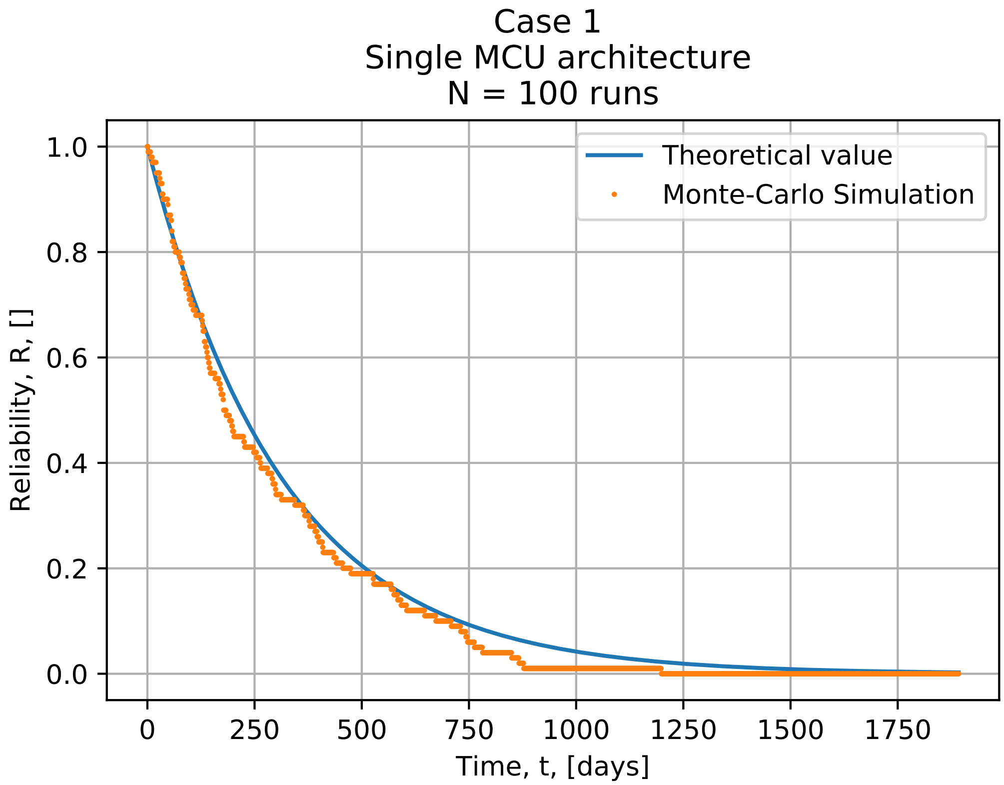
Let us know implement a Monte Carlo simulation and compare it to the expected theoretical reliability.

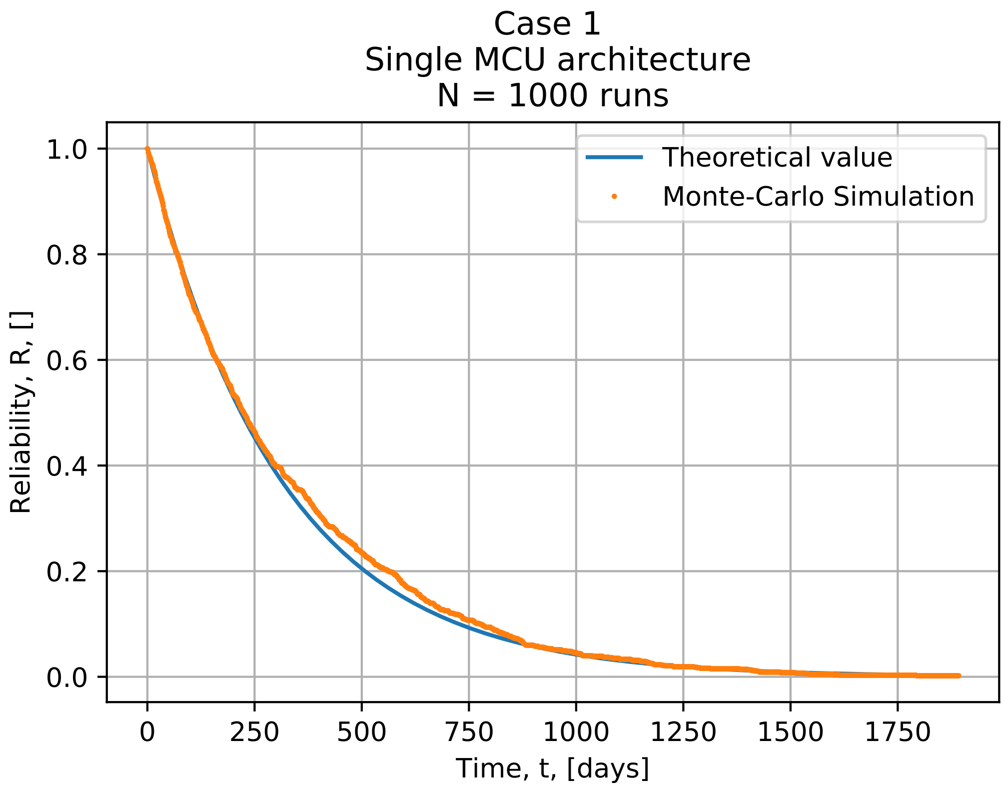
[2]:
```python
import numpy as np
import matplotlib.pyplot as plt
from random_number_generators import linear_congruential_generator

# Reliability function definition
def reliability_function(time, mttf):
    return np.exp(-time / mttf)


def single_component(runs):
    # Input data
    lambda_MCU = 3.17 * pow(10, -3)  # [per day, OMERE data]
    MTTF_MCU = 1 / lambda_MCU  # [days]

    # Define time domain

    t_start = 0
    t_end = 6 * MTTF_MCU
    t_step = 1
    time_domain = np.arange(t_start, t_end, t_step)
    random_numbers = linear_congruential_generator(0.5, runs+1)

    # Calculate theoretical reliability at each time step
    theoretical_reliability = []
```

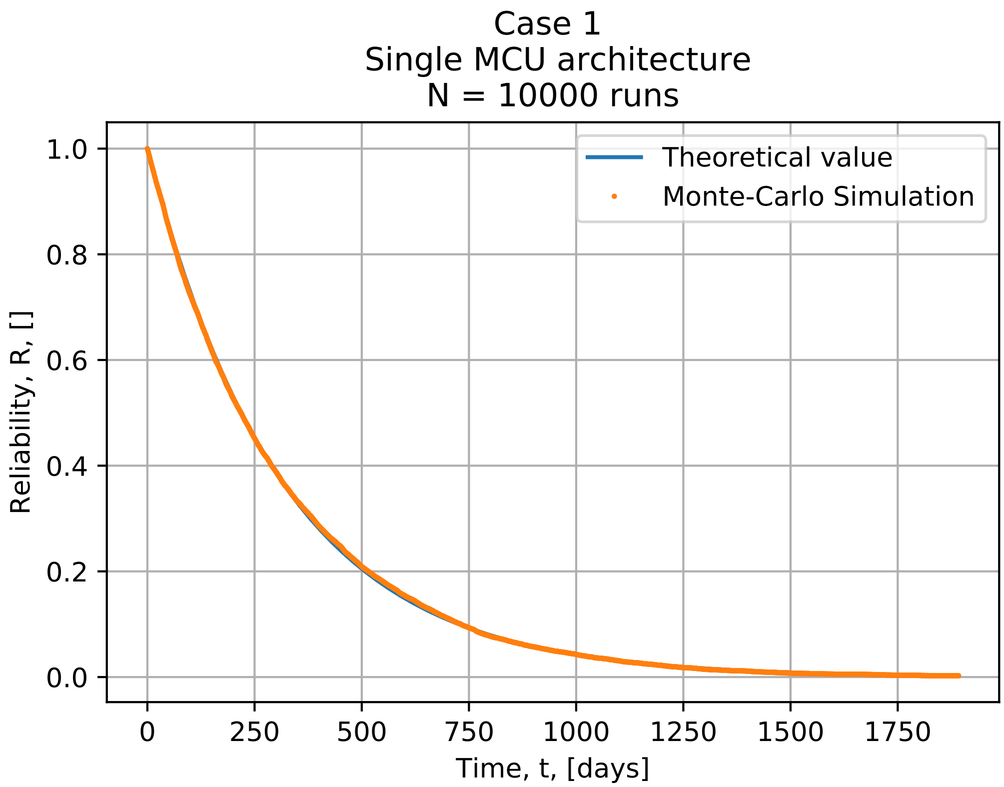
```
    for i in range(0, np.size(time_domain)):
        theoretical_reliability.append(reliability_function(time_domain[i],␣
↪MTTF_MCU))

    # Main Monte-Carlo Simulation
    failure_time = []
    working_devices = np.zeros(np.size(time_domain))
    for i in range(0, runs):
        random_value = random_numbers[i+1]
        for j in range(0, np.size(time_domain)):
            if theoretical_reliability[j] < random_value:
                failure_time.append(time_domain[j])
                break
            working_devices[j] += 1

    plt.figure()
    plt.plot(time_domain, theoretical_reliability, label="Theoretical value")
    plt.plot(time_domain, working_devices / runs, '.', markersize=2,␣
↪label="Monte-Carlo Simulation")
    plt.xlabel("Time, t, [days]")
    plt.ylabel("Reliability, R, []")
    plt.title("Case 1 \n Single MCU architecture\nN = %d runs" % runs)
    plt.grid()
    plt.legend()
    plt.show()
    return (working_devices / runs)


# Number of experiment runs
runs = [1, 10, 100, 1000, 10000]
for i in range(0, np.size(runs)):
    result_case1 = single_component(runs[i])
```


Case 1
Single MCU architecture
N = 1 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 1
Single MCU architecture
N = 10 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
Time, t, [days]
0.0
0.2
0.4
0.6
0.8
1.0
0
250
500
750
1000
1250
1500
1750

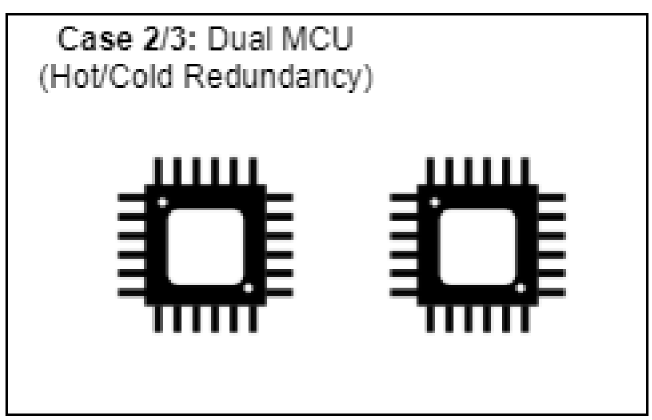

Case 1
Single MCU architecture
N = 100 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
Time, t, [days]
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750


Case 1
Single MCU architecture
N = 1000 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]



1.6 Case 2: Dual MCU (Hot Redundancy)

The architecture is the following:



The probability that we have a functional system is:

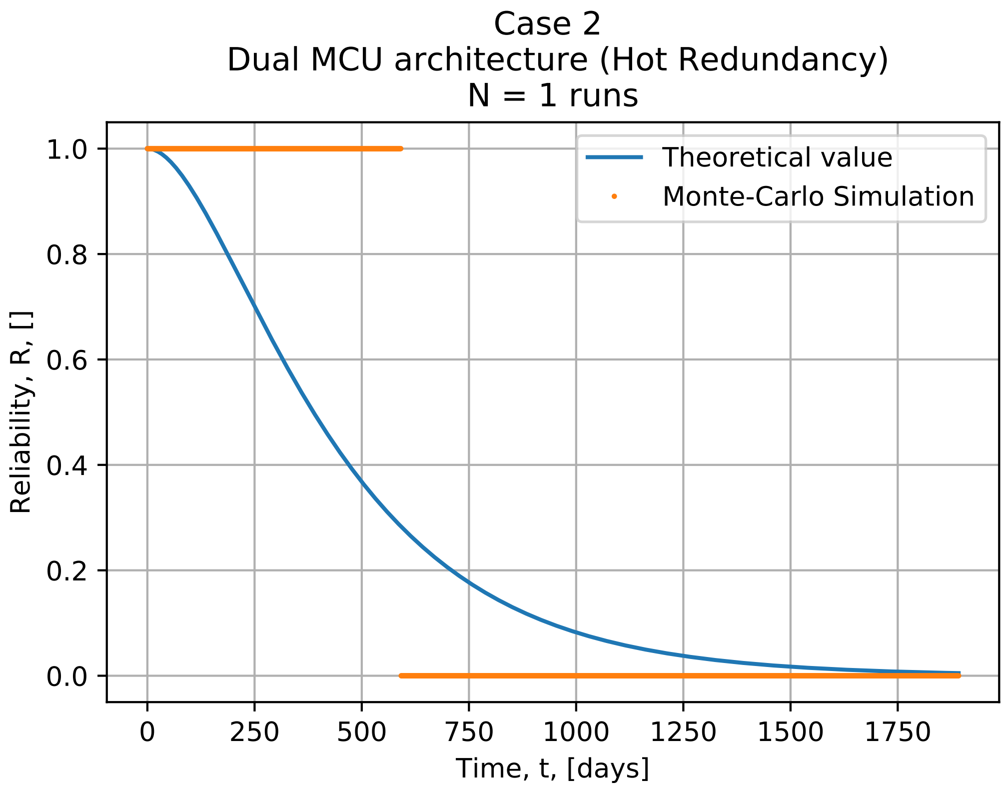
$$P_{\text{sys}} = (P_{MCU1} \cup P_{MCU2})$$

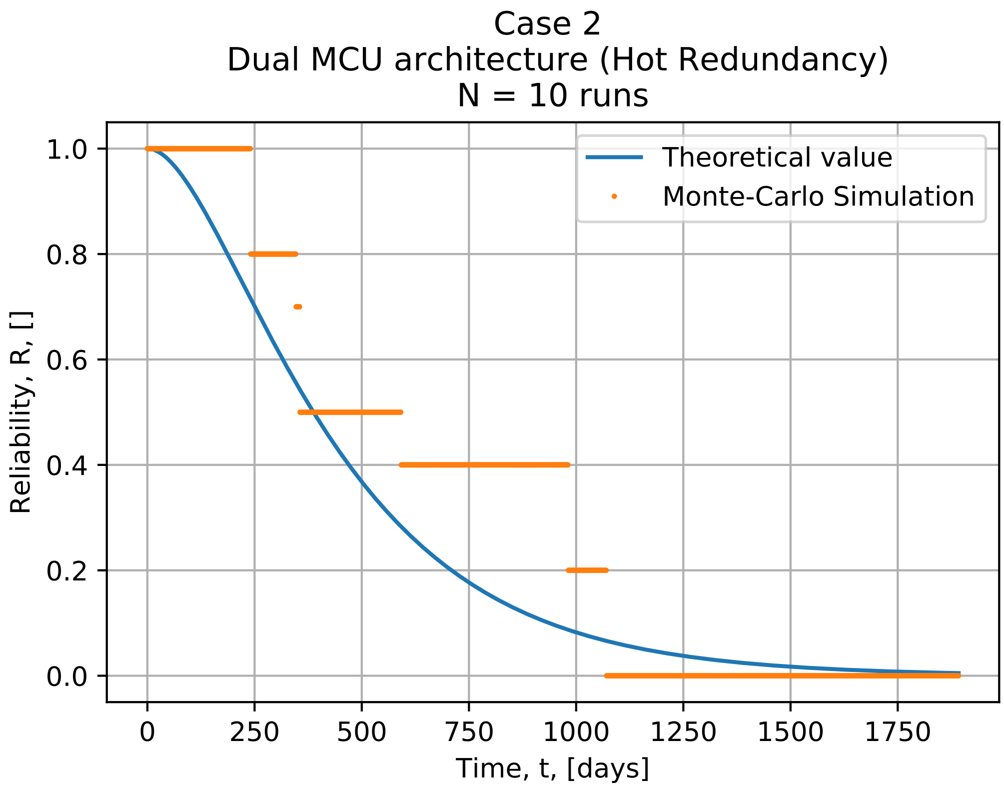
or in terms of reliability:

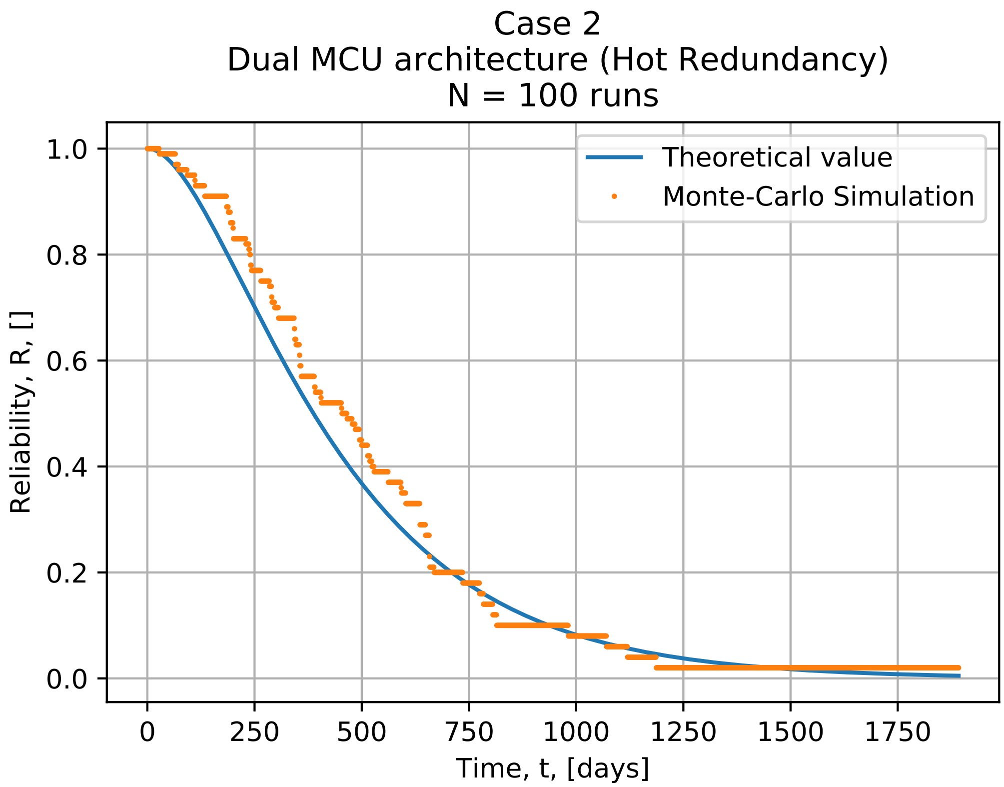
$$R_{sys} = (1 - (1 - R_{MCU1})(1 - R_{MCU2}))$$

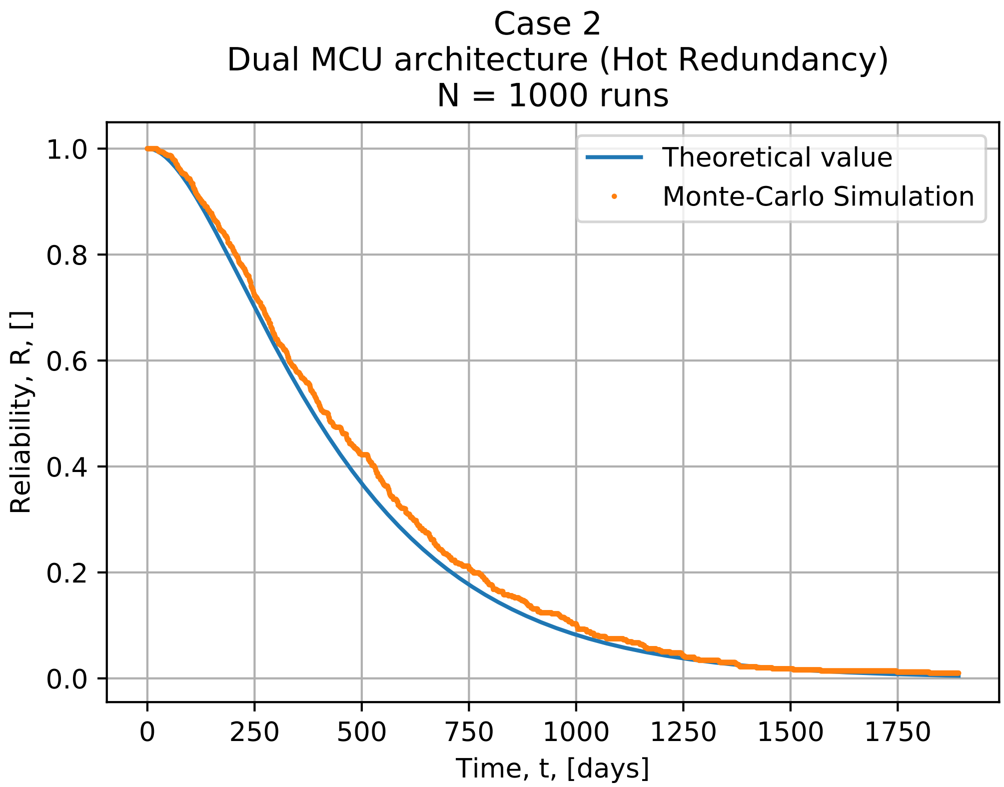
[3]:
```
import numpy as np
import matplotlib.pyplot as plt
from random_number_generators import linear_congruential_generator

# Reliability function definition
def system_reliability_function(time, mttf):
    return 1-((1-np.exp(-time / mttf))*(1-np.exp(-time / mttf)))


# Component reliability
def component_reliability_function(time, mttf):
    return np.exp(-time / mttf)


def dual_mcu(runs):
    # Input data
    lambda_MCU = 3.17 * pow(10, -3)  # [per day, OMERE data]
    MTTF_MCU = 1 / lambda_MCU  # [days]

    # Define time domain

    t_start = 0
    t_end = 6 * MTTF_MCU
    t_step = 1
    time_domain = np.arange(t_start, t_end, t_step)
    random_numbers = linear_congruential_generator(0.862, runs+2)

    # Calculate theoretical reliability at each time step
    theoretical_system_reliability = []
    theoretical_component_reliability = []
    for i in range(0, np.size(time_domain)):
        theoretical_system_reliability.
↪append(system_reliability_function(time_domain[i], MTTF_MCU))
        theoretical_component_reliability.
↪append(component_reliability_function(time_domain[i], MTTF_MCU))

    # Main Monte-Carlo Simulation
    failure_time_system = []
    working_devices = np.zeros(np.size(time_domain))
    for i in range(0, runs):
        random_value_first = random_numbers[i+1]
        random_value_second = random_numbers[i+2]
        state_first_mcu = 1
        state_second_mcu = 1
```

```
        for j in range(0, np.size(time_domain)):
            if theoretical_component_reliability[j] < random_value_first:
                state_first_mcu = 0
            if theoretical_component_reliability[j] < random_value_second:
                state_second_mcu = 0
            if state_first_mcu == 0 and state_second_mcu == 0:
                failure_time_system.append(time_domain[j])
                break
            working_devices[j] += 1

    plt.figure()
    plt.plot(time_domain, theoretical_system_reliability, label="Theoretical␣
↪value")
    plt.plot(time_domain, working_devices / runs, '.', markersize=2,␣
↪label="Monte-Carlo Simulation")
    plt.xlabel("Time, t, [days]")
    plt.ylabel("Reliability, R, []")
    plt.title("Case 2 \n Dual MCU architecture (Hot Redundancy)\nN = %d runs" %␣
↪runs)
    plt.grid()
    plt.legend()
    plt.show()
    return (working_devices / runs)


# Number of experiment runs
runs = [1, 10, 100, 1000, 10000]
for i in range(0, np.size(runs)):
    result_case2 = dual_mcu(runs[i])
```

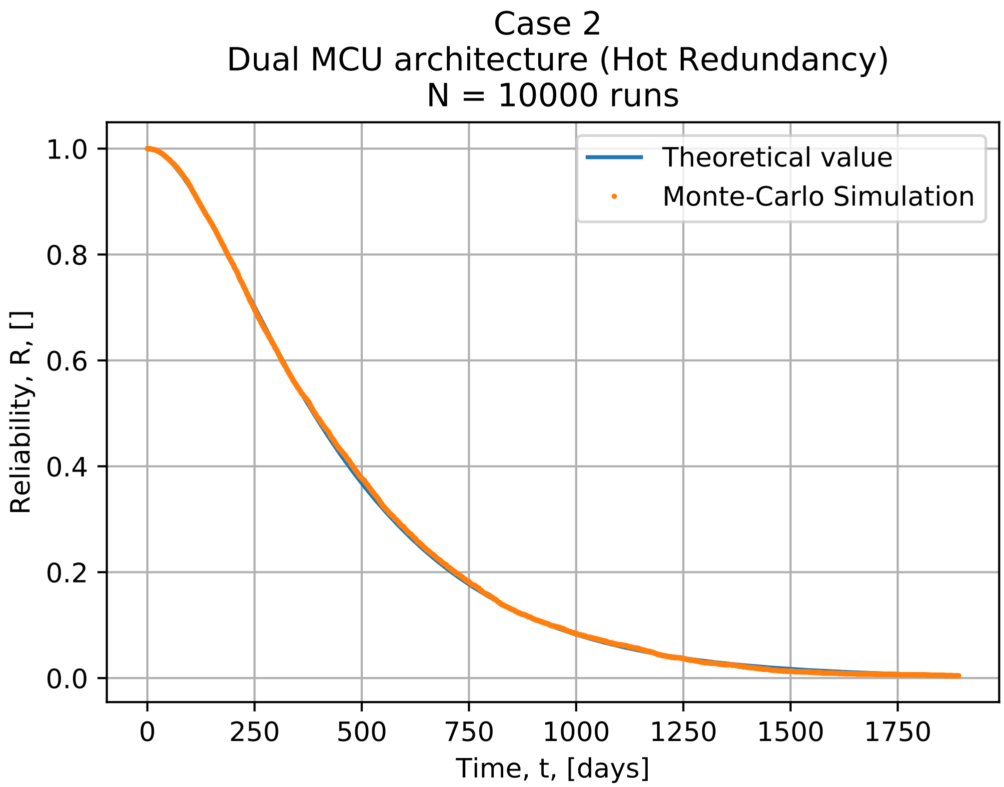

Case 2
Dual MCU architecture (Hot Redundancy)
N = 1 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 2
Dual MCU architecture (Hot Redundancy)
N = 10 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 2
Dual MCU architecture (Hot Redundancy)
N = 100 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]

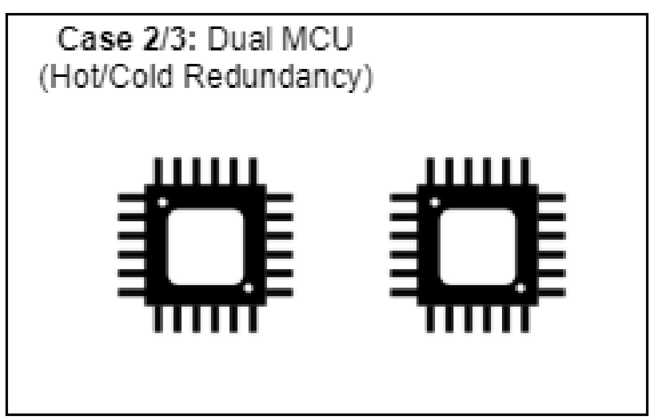


Case 2
Dual MCU architecture (Hot Redundancy)
N = 1000 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]



1.7 Case 3: Dual MCU (Cold Redundancy)

The architecture is the following:

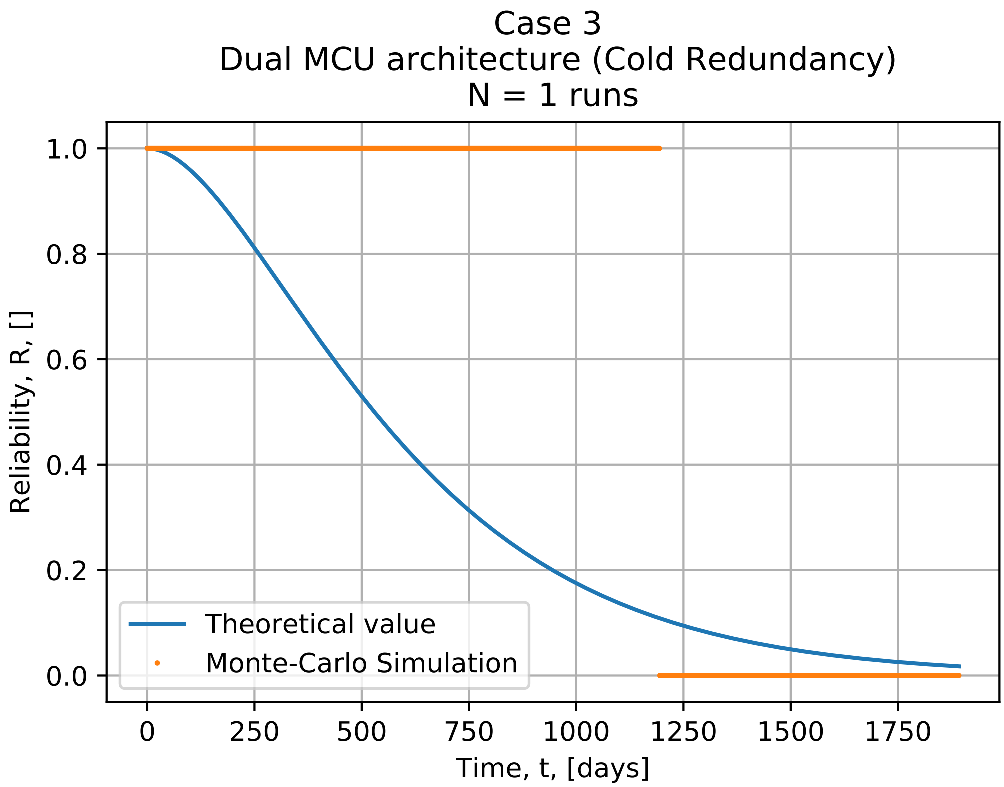


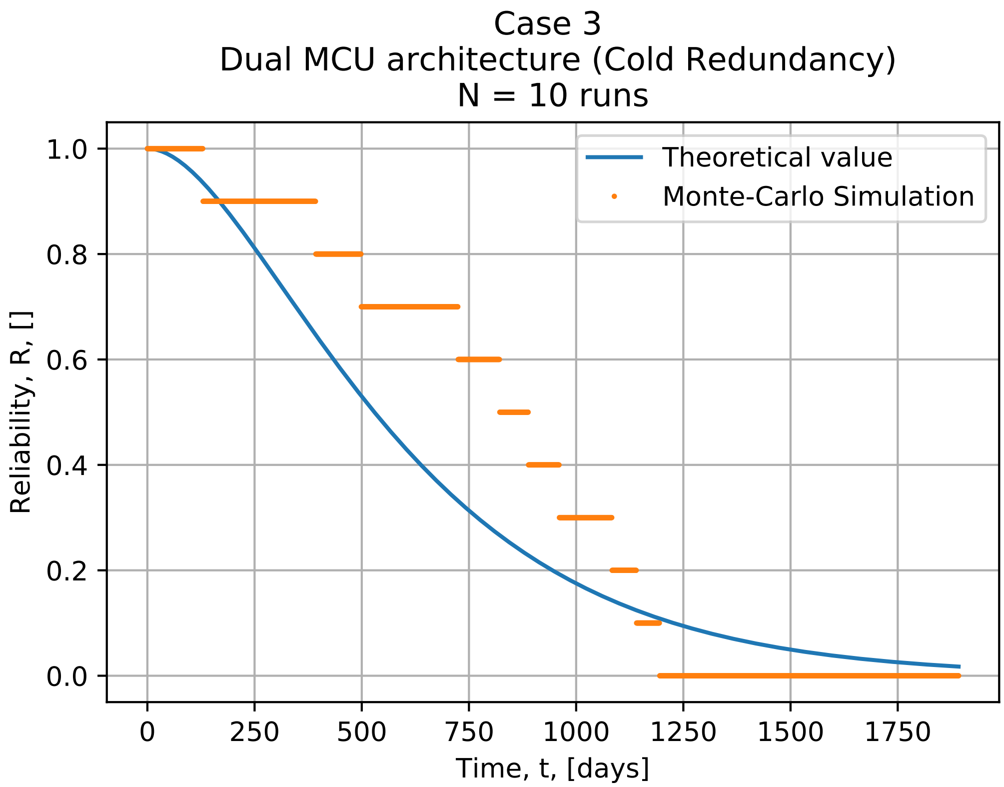
For cold redundancy, it is assumed that the second MCU is not susceptible to radiation until it starts functioning. In terms of reliability, this is calculated as:

$$R_{sys} = R_{MCU1} + R_{MCU2}\lambda_{MCU2}t$$

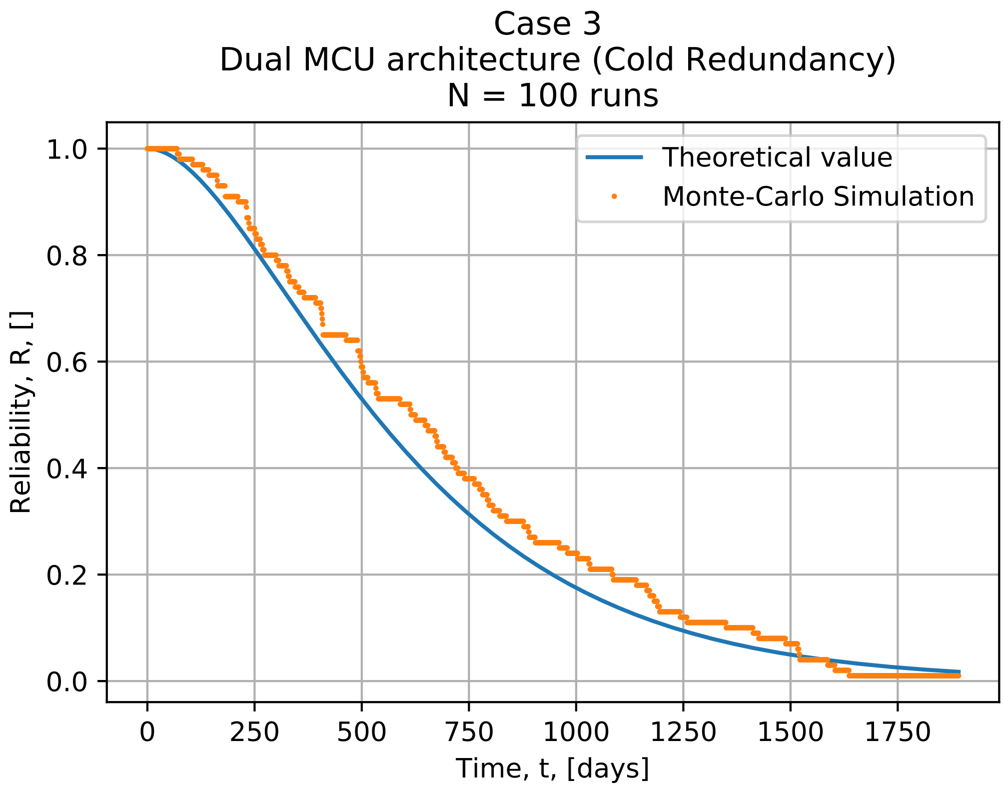
[4]:
```
import numpy as np
import matplotlib.pyplot as plt
from random_number_generators import linear_congruential_generator

# Reliability function definition
def system_reliability_function(time, mttf):
    return (np.exp(-time / mttf))+(np.exp(-time / mttf))*(time / mttf)


# Component reliability
def component_reliability_function(time, mttf):
    return np.exp(-time / mttf)

def dual_mcu(runs):
    random_numbers = linear_congruential_generator(0.742, runs+2)
    # Input data
    lambda_MCU = 3.17 * pow(10, -3)  # [per day, OMERE data]
    MTTF_MCU = 1 / lambda_MCU  # [days]

    # Define time domain

    t_start = 0
    t_end = 6 * MTTF_MCU
    t_step = 1
    time_domain = np.arange(t_start, t_end, t_step)

    # Calculate theoretical reliability at each time step
    theoretical_system_reliability = []
    theoretical_component_reliability = []
    for i in range(0, np.size(time_domain)):
        theoretical_system_reliability.
↪append(system_reliability_function(time_domain[i], MTTF_MCU))
        theoretical_component_reliability.
↪append(component_reliability_function(time_domain[i], MTTF_MCU))

    # Main Monte-Carlo Simulation
    failure_time_system = []
    working_systems = np.zeros(np.size(time_domain))
    for i in range(0, runs):
        random_value_first = random_numbers[i+1]
        random_value_second = random_numbers[i+2]
        state_first_mcu = 1
        state_second_mcu = 1
        for j in range(0, np.size(time_domain)):
            if theoretical_component_reliability[j] < random_value_first:
                for k in range(j, np.size(time_domain)):
```

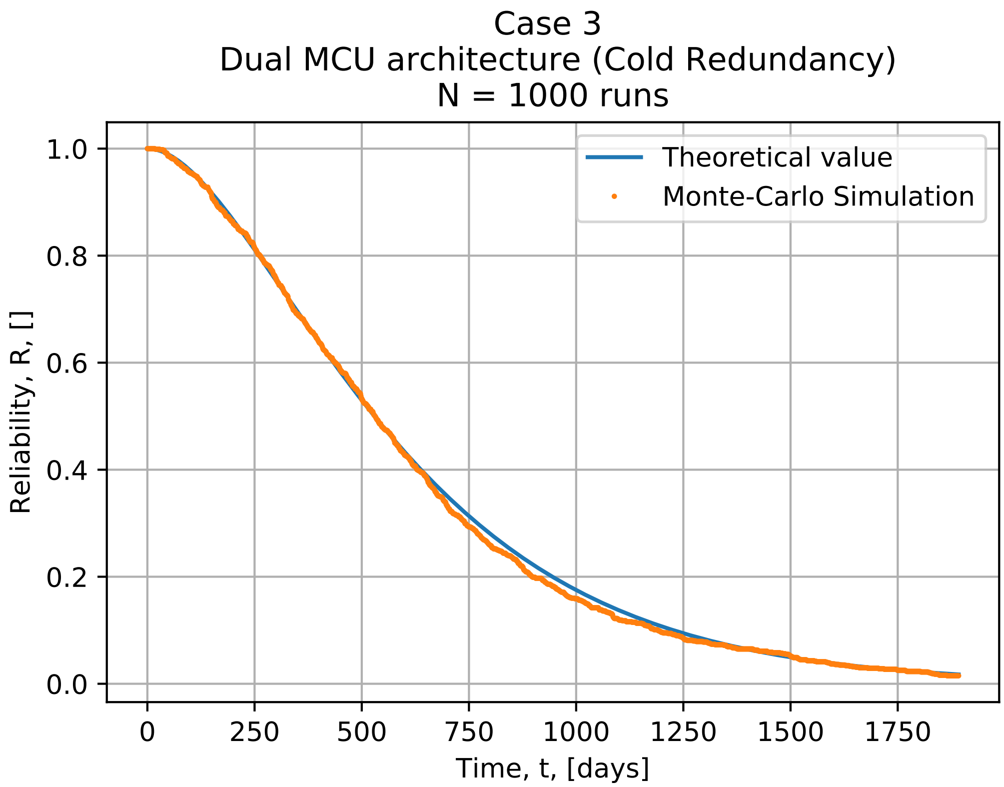
```
                    if theoretical_component_reliability[k-j] <␣
↪random_value_second:
                        failure_time_system.append(time_domain[k])
                        break
                    working_systems[k] += 1
                break
            working_systems[j] += 1

    plt.figure()
    plt.plot(time_domain, theoretical_system_reliability, label="Theoretical␣
↪value")
    plt.plot(time_domain, working_systems / runs, '.', markersize=2,␣
↪label="Monte-Carlo Simulation")
    plt.xlabel("Time, t, [days]")
    plt.ylabel("Reliability, R, []")
    plt.title("Case 3 \n Dual MCU architecture (Cold Redundancy)\nN = %d runs"␣
↪% runs)
    plt.grid()
    plt.legend()
    plt.show()
    return (working_systems / runs)


# Number of experiment runs
runs = [1, 10, 100, 1000, 10000]
for i in range(0, np.size(runs)):
    result_case3 = dual_mcu(runs[i])
```


Case 3
Dual MCU architecture (Cold Redundancy)
N = 1 runs
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]
Theoretical value
Monte-Carlo Simulation

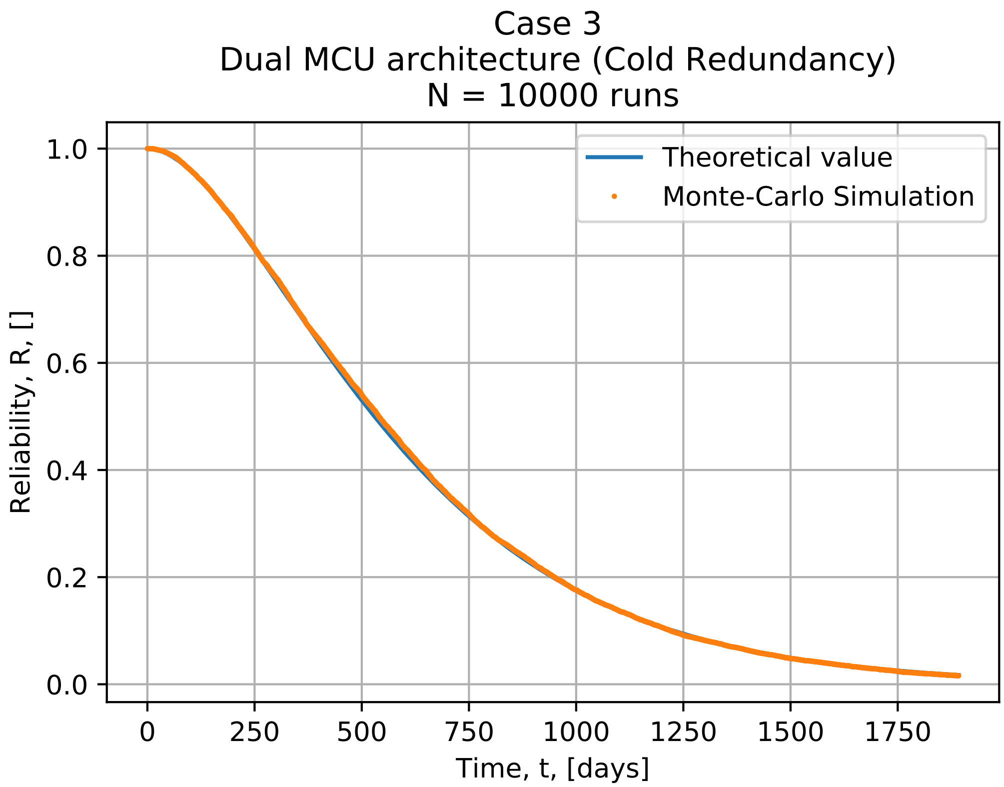


Case 3
Dual MCU architecture (Cold Redundancy)
N = 10 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 3
Dual MCU architecture (Cold Redundancy)
N = 100 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
Time, t, [days]
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750

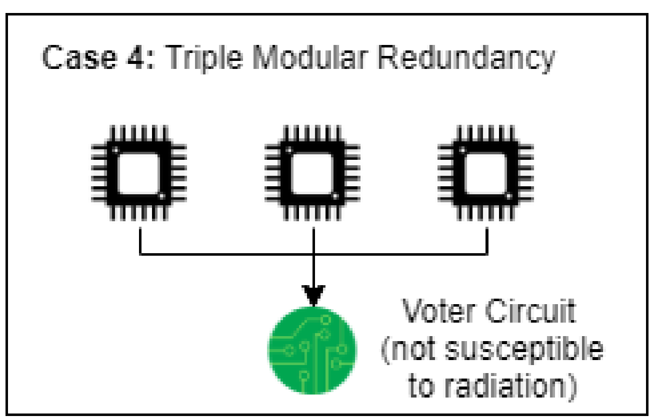


Case 3
Dual MCU architecture (Cold Redundancy)
N = 1000 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
Time, t, [days]
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750



1.8 Case 4: Triple modular redundancy (Immune voter)

The architecture is the following:



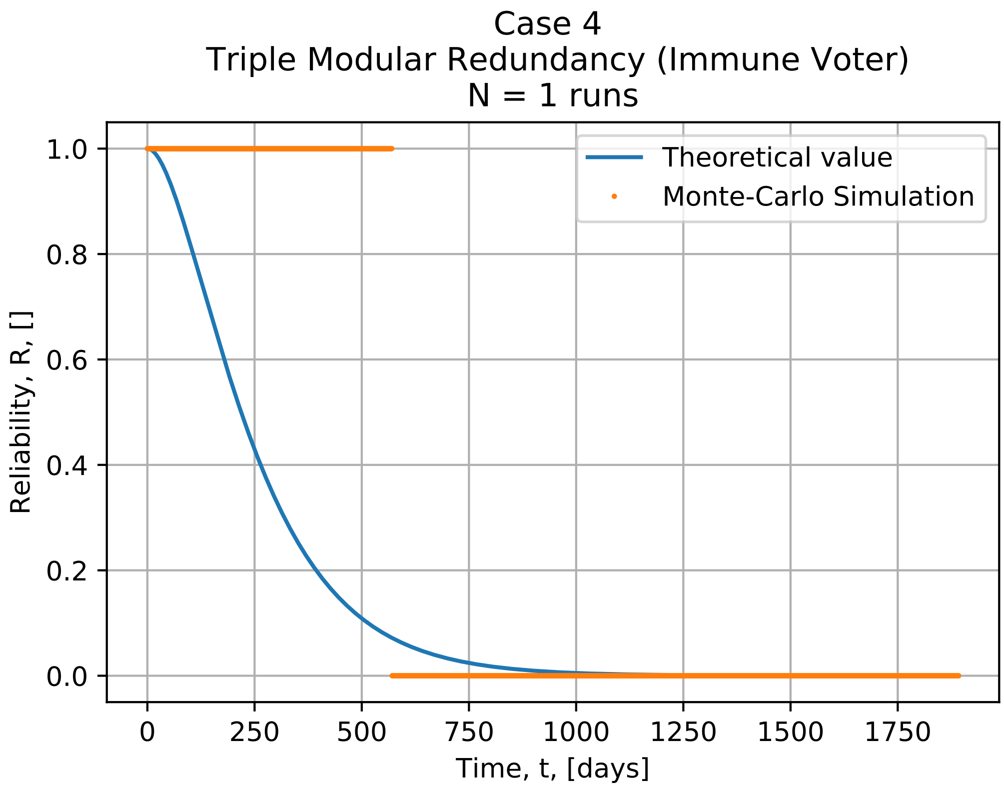
the reliability function is given by:

$$\mathrm{R_{sys}} = \mathrm{R_{voter}} \left(3\mathrm{R}_{\mathrm{MCU}}^{2} - 2\mathrm{R}_{\mathrm{MCU}}^{3}\right)$$

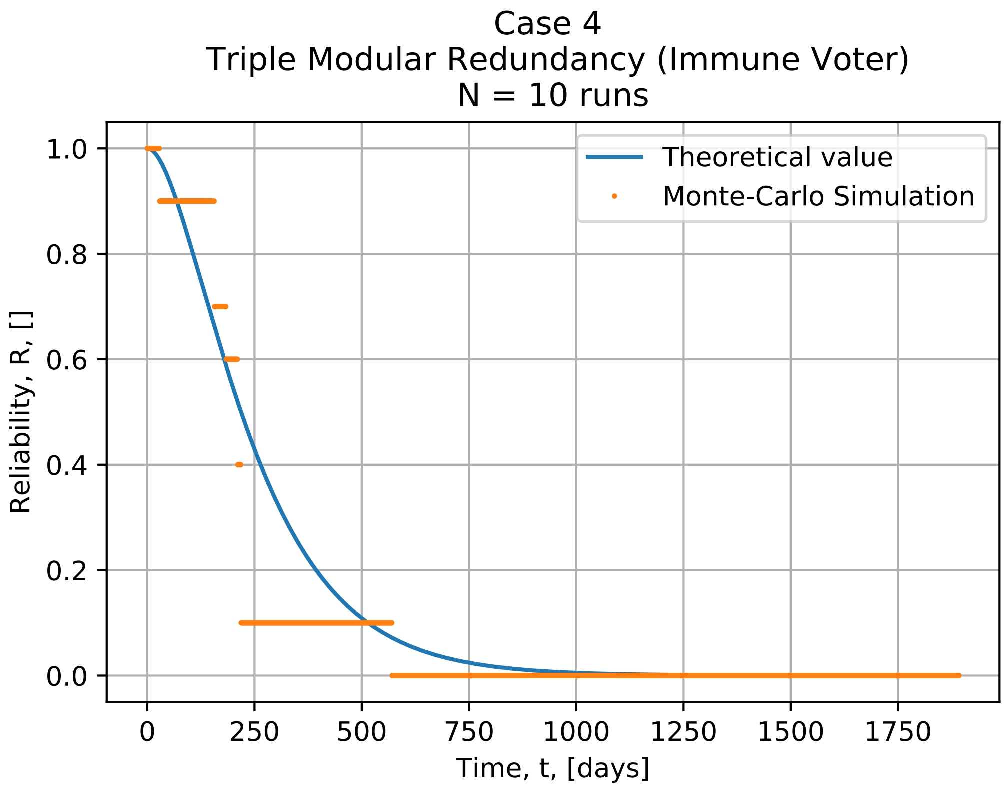
where $R_{voter} = 1$ since the voter is considered to be immune to radiation.

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from random_number_generators import linear_congruential_generator


# Reliability function definition
def system_reliability_function(time, mttf):
    return 3*pow(np.exp(-time / mttf), 2) - 2*pow(np.exp(-time / mttf), 3)


# Component reliability
def component_reliability_function(time, mttf):
    return np.exp(-time / mttf)


def tmr(runs):
    # Input data
    lambda_MCU = 3.17 * pow(10, -3)  # [per day, OMERE data]
    MTTF_MCU = 1 / lambda_MCU  # [days]

    # Define time domain

    t_start = 0
    t_end = 6 * MTTF_MCU
    t_step = 1
    time_domain = np.arange(t_start, t_end, t_step)
    random_numbers = linear_congruential_generator(0.389, runs+3)

    # Calculate theoretical reliability at each time step
    theoretical_system_reliability = []
    theoretical_component_reliability = []
    for i in range(0, np.size(time_domain)):
        theoretical_system_reliability.
↪append(system_reliability_function(time_domain[i], MTTF_MCU))
        theoretical_component_reliability.
↪append(component_reliability_function(time_domain[i], MTTF_MCU))

    # Main Monte-Carlo Simulation
    failure_time_system = []
    working_systems = np.zeros(np.size(time_domain))
    for i in range(0, runs):
        random_value_first = random_numbers[i+1]
        random_value_second = random_numbers[i+2]
        random_value_third = random_numbers[i+3]
        state_first_mcu = 1
        state_second_mcu = 1
        state_third_mcu = 1
```

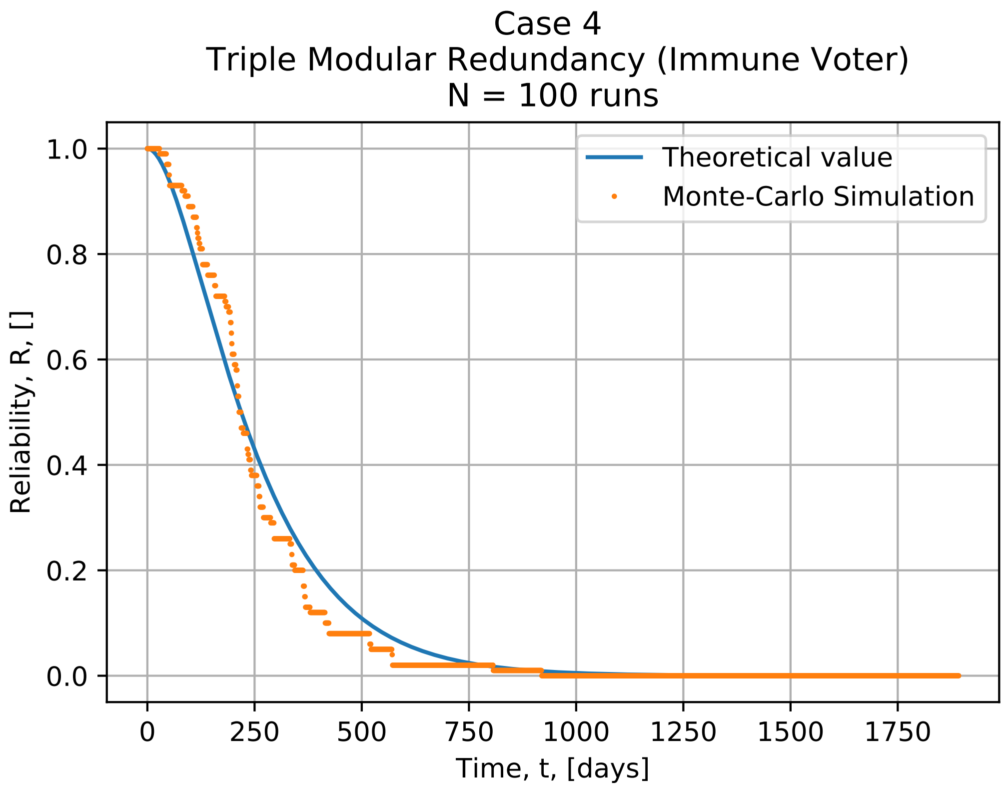
```
        for j in range(0, np.size(time_domain)):
            if theoretical_component_reliability[j] < random_value_first:
                state_first_mcu = 0
            if theoretical_component_reliability[j] < random_value_second:
                state_second_mcu = 0
            if theoretical_component_reliability[j] < random_value_third:
                state_third_mcu = 0
            total_state = state_first_mcu + state_second_mcu + state_third_mcu
            if total_state < 2:
                failure_time_system.append(time_domain[j])
                break
            working_systems[j] += 1

    plt.figure()
    plt.plot(time_domain, theoretical_system_reliability, label="Theoretical␣
↪value")
    plt.plot(time_domain, working_systems / runs, '.', markersize=2,␣
↪label="Monte-Carlo Simulation")
    plt.xlabel("Time, t, [days]")
    plt.ylabel("Reliability, R, []")
    plt.title("Case 4 \n Triple Modular Redundancy (Immune Voter)\nN = %d runs"␣
↪% runs)
    plt.grid()
    plt.legend()
    plt.show()
    return (working_systems / runs)


# Number of experiment runs
runs = [1, 10, 100, 1000, 10000]
for i in range(0, np.size(runs)):
    result_case4 = tmr(runs[i])
```



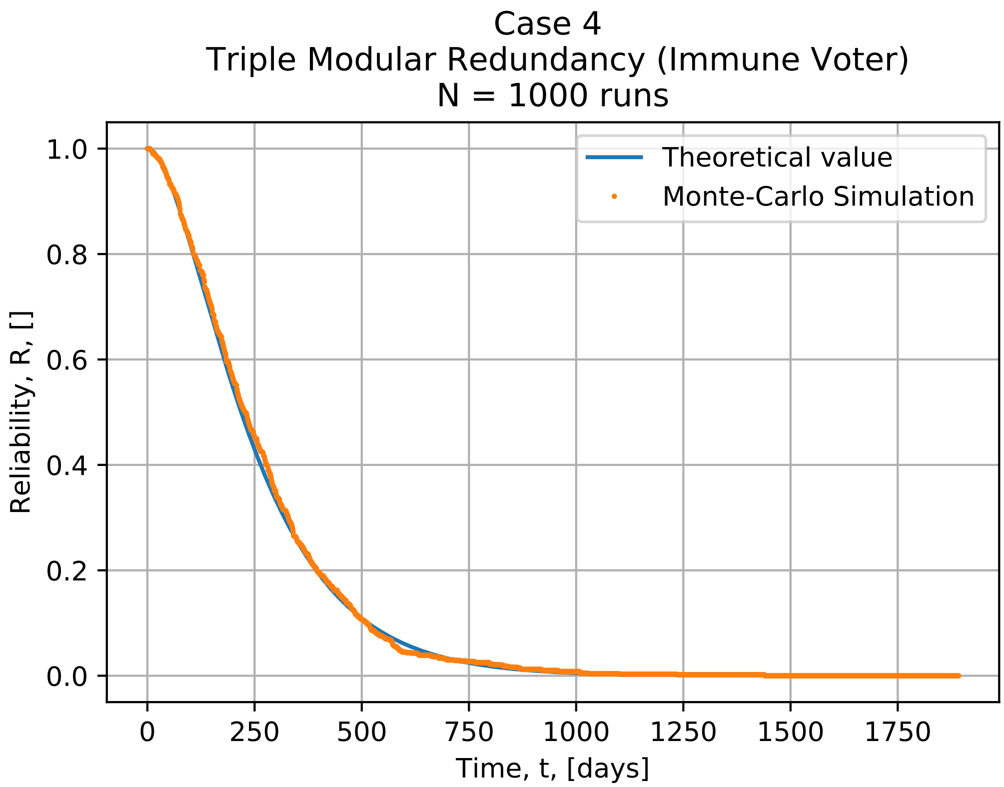
Case 4
Triple Modular Redundancy (Immune Voter)
N = 1 runs
Theoretical value
Monte-Carlo Simulation
1.0
0.8
0.6
0.4
0.2
0.0
Reliability, R, []
0
250
500
750
1000
1250
1500
1750
Time, t, [days]

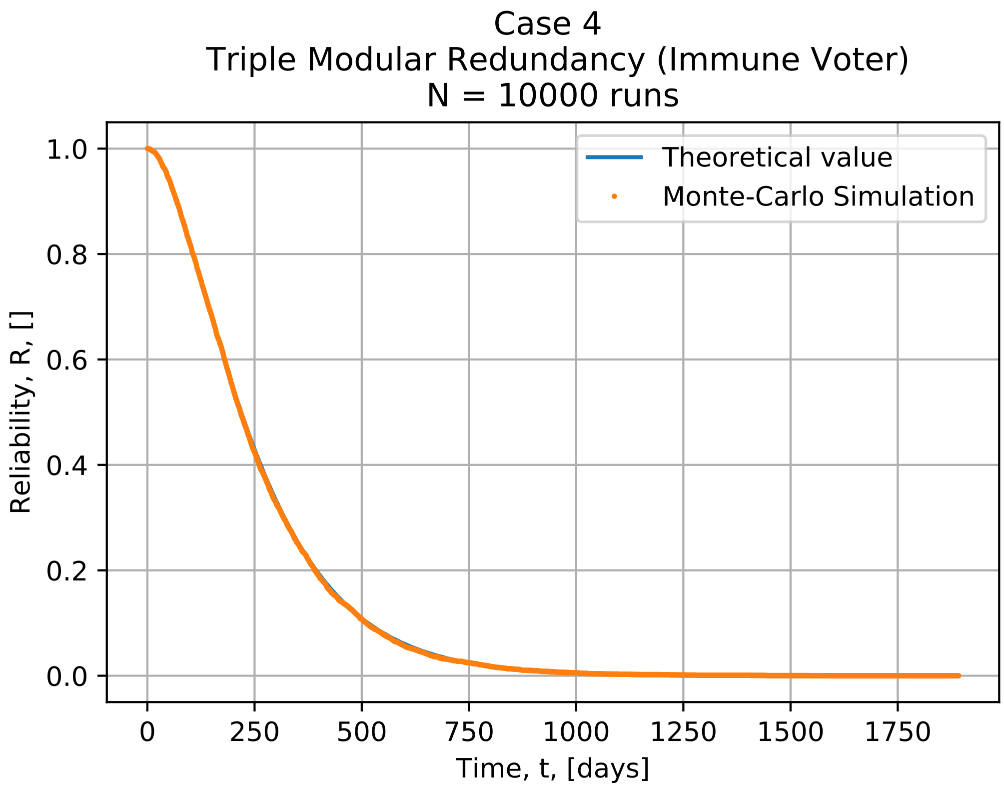


Case 4
Triple Modular Redundancy (Immune Voter)
N = 10 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]



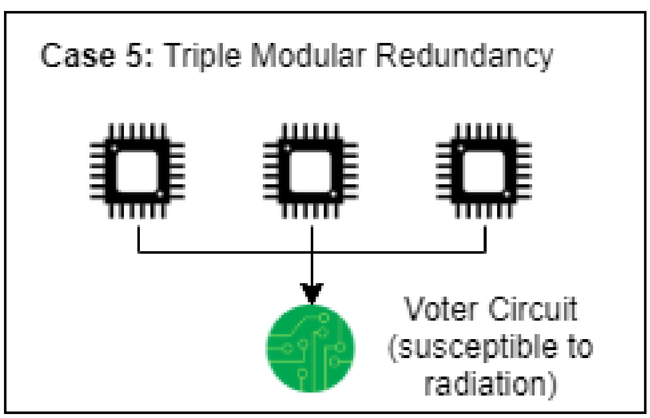
Case 4
Triple Modular Redundancy (Immune Voter)
N = 100 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 4
Triple Modular Redundancy (Immune Voter)
N = 1000 runs
Theoretical value
Monte-Carlo Simulation
1.0
0.8
0.6
0.4
0.2
0.0
Reliability, R, []
0
250
500
750
1000
1250
1500
1750
Time, t, [days]



1.9 Case 5: Triple modular redundancy (Vulnerable voter)

The architecture is the following:



For this case the reliability function is given by:

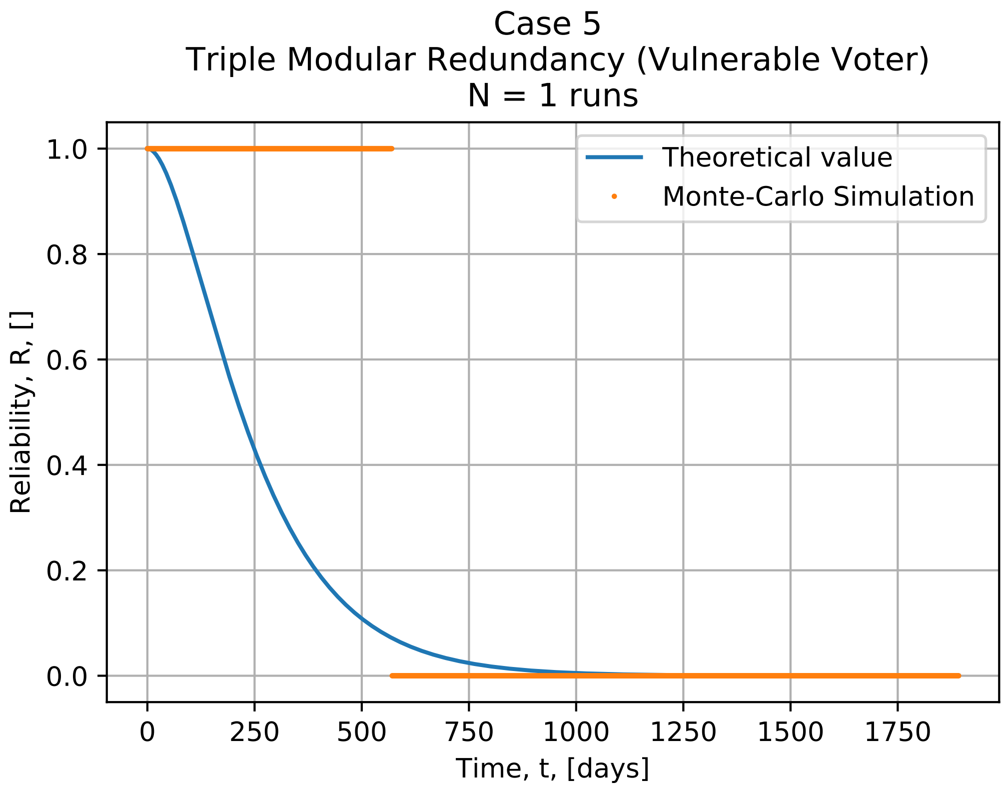
$$\mathrm{R_{sys}} = \mathrm{R_{voter}}\left(3\mathrm{R}_{\mathrm{MCU}}^{2} - 2\mathrm{R}_{\mathrm{MCU}}^{3}\right)$$

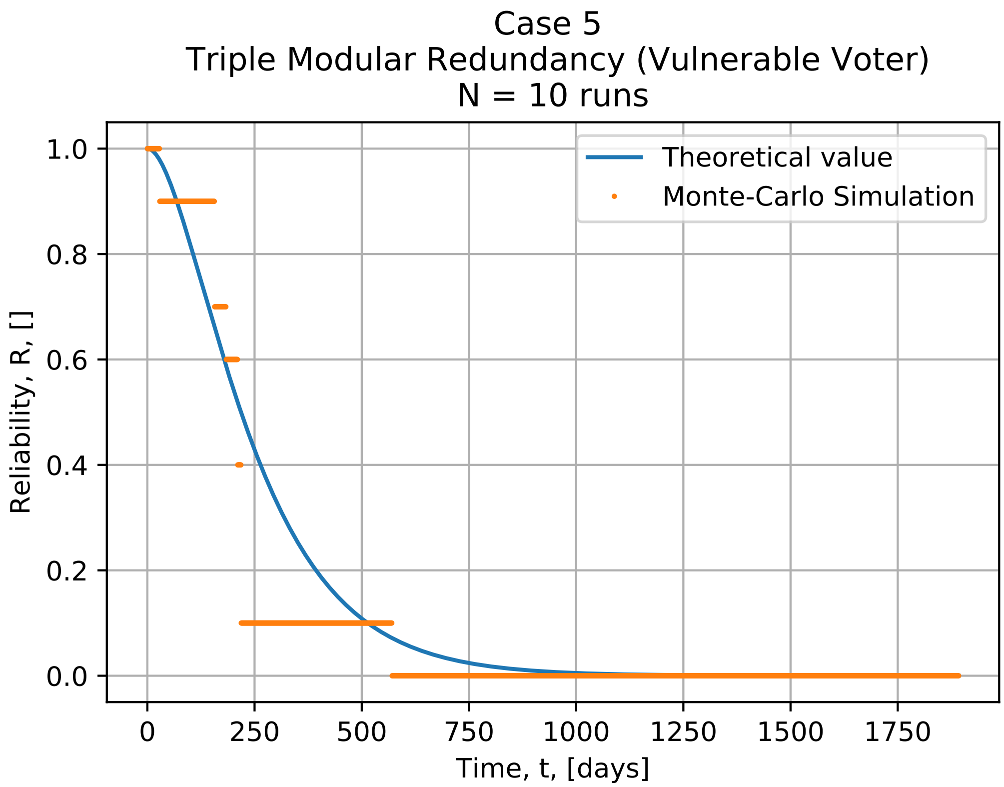
where R_{voter} is now different than one (computed using the failure rate for the voter).

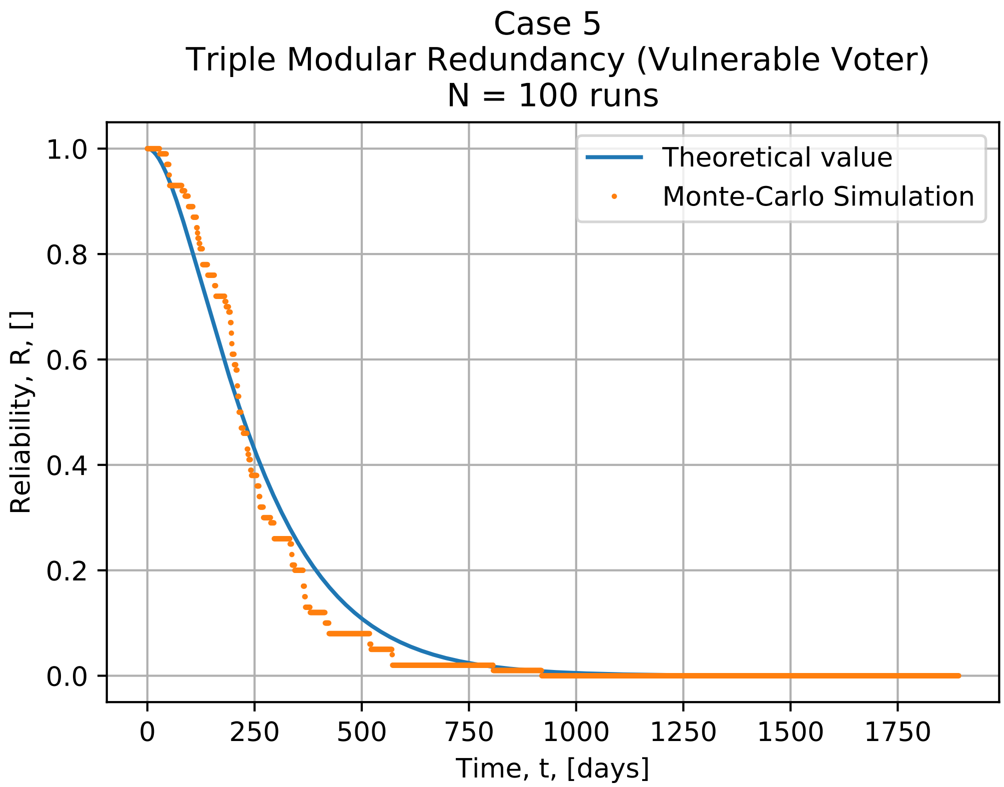
[6]:

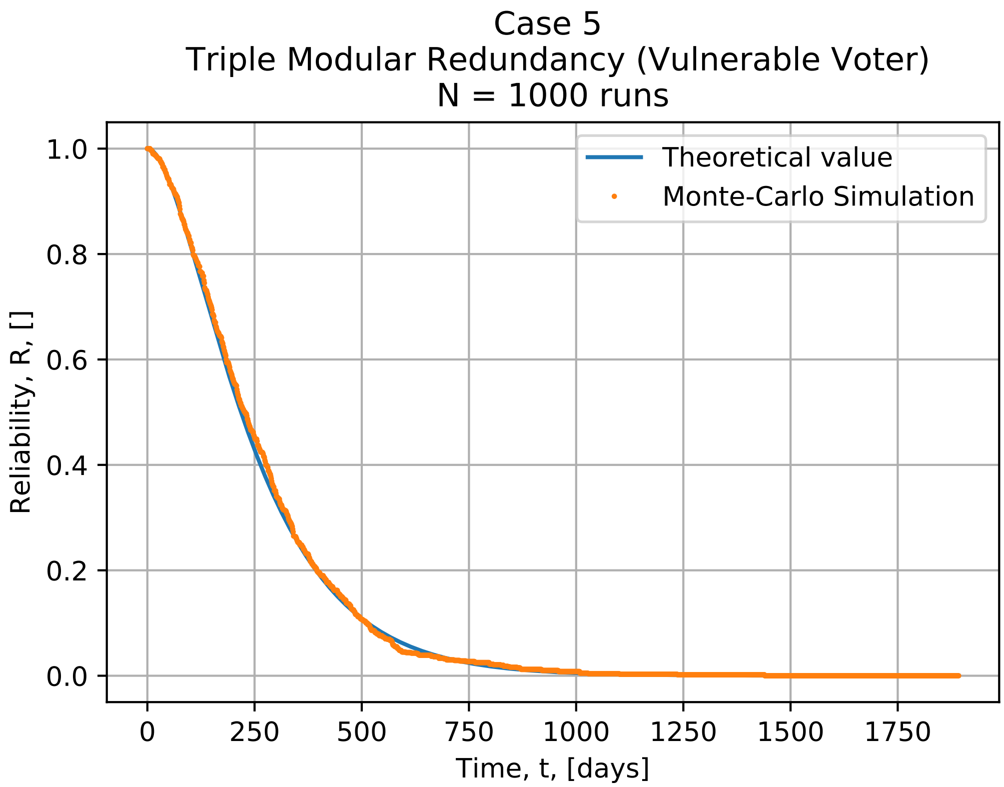
```
import numpy as np
import matplotlib.pyplot as plt
from random_number_generators import linear_congruential_generator

# Reliability function definition
def system_reliability_function(time, mttf_mcu, mttf_voter):
    return np.exp(-time / mttf_voter)*(3*pow(np.exp(-time / mttf_mcu), 2) -␣
↪2*pow(np.exp(-time / mttf_mcu), 3))


# Component reliability
def component_reliability_function(time, mttf):
    return np.exp(-time / mttf)


def tmr(runs):
    random_numbers = linear_congruential_generator(0.389, runs+4)
    # Input data
    lambda_MCU = 3.17 * pow(10, -3)  # [per day, OMERE data]
    MTTF_MCU = 1 / lambda_MCU  # [days]

    lambda_voter = 6.02 * pow(10, -6)  # [per day, OMERE data]
    MTTF_voter = 1 / lambda_voter  # [days]

    # Define time domain

    t_start = 0
    t_end = 6 * MTTF_MCU
    t_step = 1
    time_domain = np.arange(t_start, t_end, t_step)

    # Calculate theoretical reliability at each time step
    theoretical_system_reliability = []
    theoretical_component_reliability = []
    theoretical_voter_reliability = []
    for i in range(0, np.size(time_domain)):
        theoretical_system_reliability.
↪append(system_reliability_function(time_domain[i], MTTF_MCU, MTTF_voter))
        theoretical_component_reliability.
↪append(component_reliability_function(time_domain[i], MTTF_MCU))
        theoretical_voter_reliability.
↪append(component_reliability_function(time_domain[i], MTTF_voter))

    # Main Monte-Carlo Simulation
    failure_time_system = []
    working_systems = np.zeros(np.size(time_domain))
    for i in range(0, runs):
```

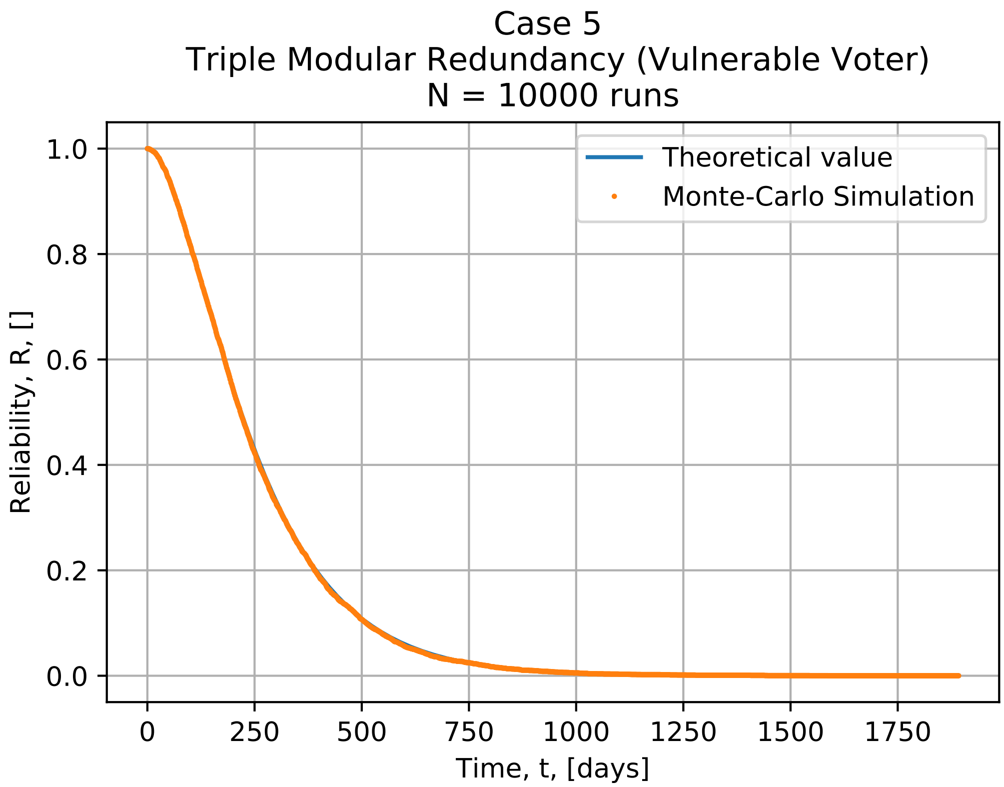
```
        random_value_first = random_numbers[i+1]
        random_value_second = random_numbers[i+2]
        random_value_third = random_numbers[i+3]
        random_value_voter = random_numbers[i+4]
        state_first_mcu = 1
        state_second_mcu = 1
        state_third_mcu = 1
        state_voter = 1
        for j in range(0, np.size(time_domain)):
            if theoretical_component_reliability[j] < random_value_first:
                state_first_mcu = 0
            if theoretical_component_reliability[j] < random_value_second:
                state_second_mcu = 0
            if theoretical_component_reliability[j] < random_value_third:
                state_third_mcu = 0
            if theoretical_voter_reliability[j] < random_value_voter:
                state_voter = 0
            total_state = state_first_mcu + state_second_mcu + state_third_mcu
            if total_state < 2 or state_voter == 0:
                failure_time_system.append(time_domain[j])
                break
            working_systems[j] += 1

    plt.figure()
    plt.plot(time_domain, theoretical_system_reliability, label="Theoretical␣
↪value")
    plt.plot(time_domain, working_systems / runs, '.', markersize=2,␣
↪label="Monte-Carlo Simulation")
    plt.xlabel("Time, t, [days]")
    plt.ylabel("Reliability, R, []")
    plt.title("Case 5 \n Triple Modular Redundancy (Vulnerable Voter)\nN = %d␣
↪runs" % runs)
    plt.grid()
    plt.legend()
    plt.show()
    return time_domain, working_systems / runs


# Number of experiment runs
runs = [1, 10, 100, 1000, 10000]
for i in range(0, np.size(runs)):
    time_domain, result_case5 = tmr(runs[i])
```


Case 5
Triple Modular Redundancy (Vulnerable Voter)
N = 1 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]


Case 5
Triple Modular Redundancy (Vulnerable Voter)
N = 10 runs
Reliability, R, []
Time, t, [days]
Theoretical value
Monte-Carlo Simulation
0.0
0.2
0.4
0.6
0.8
1.0
0
250
500
750
1000
1250
1500
1750

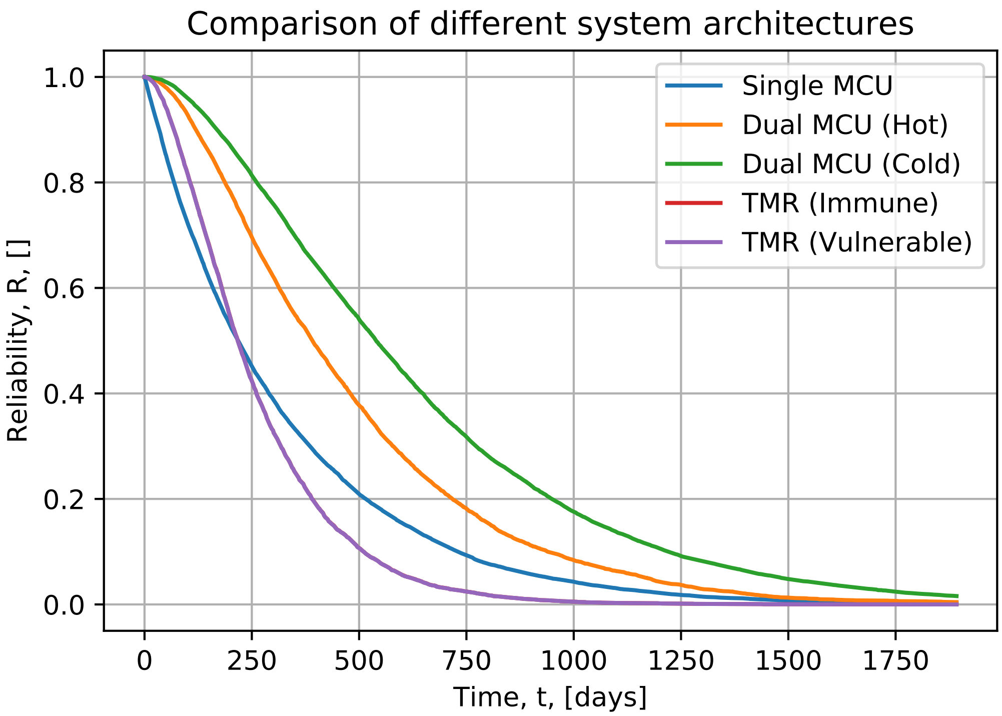

Case 5
Triple Modular Redundancy (Vulnerable Voter)
N = 100 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
Time, t, [days]
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750


Case 5
Triple Modular Redundancy (Vulnerable Voter)
N = 1000 runs
Theoretical value
Monte-Carlo Simulation
Reliability, R, []
1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
Time, t, [days]



1.10 Compare all cases

[7]:
```
plt.figure()
plt.plot(time_domain, result_case1, label="Single MCU")
plt.plot(time_domain, result_case2, label="Dual MCU (Hot)")
plt.plot(time_domain, result_case3, label="Dual MCU (Cold)")
plt.plot(time_domain, result_case4, label="TMR (Immune)")
plt.plot(time_domain, result_case5, label="TMR (Vulnerable)")
plt.grid()
plt.title("Comparison of different system architectures")
plt.xlabel("Time, t, [days]")
plt.ylabel("Reliability, R, []")
plt.legend()
plt.show()
```



1.11 Conclusions

We note that in case of destructive single event effect, the dual MCU architectures seem to be the most reliable choice out of all the examined architectures. We can also note that for around $N = 10000$ runs we are receiving the expected behavior for each system. Future work on this subject can include the implementation of non-destructive single event effects, which will highlight the advantages of the triple modular redundancy architectures for this system.

1.12 References

1. *Enrico Zio*, The Monte Carlo Simulation Method for System Reliability and Risk Analysis, Springer Verlag, 2013
2. *Stylianos Massen*, Mathematical Methods in Physics: Part II, 2014
3. *Martin L. Shooman*, Reliability of Computer Systems and Networks, Wiley, 2002
4. *Bob Gigliuto et al.*, SEE Test Report Cypress EZ-USB FX2 USB Microcontroller and SMSC USB2512 USB Hub, TAMU, 2011
5. *Texas Instruments*, Single-event Effects Test Report of the MSP430FR5969-SP, 2019