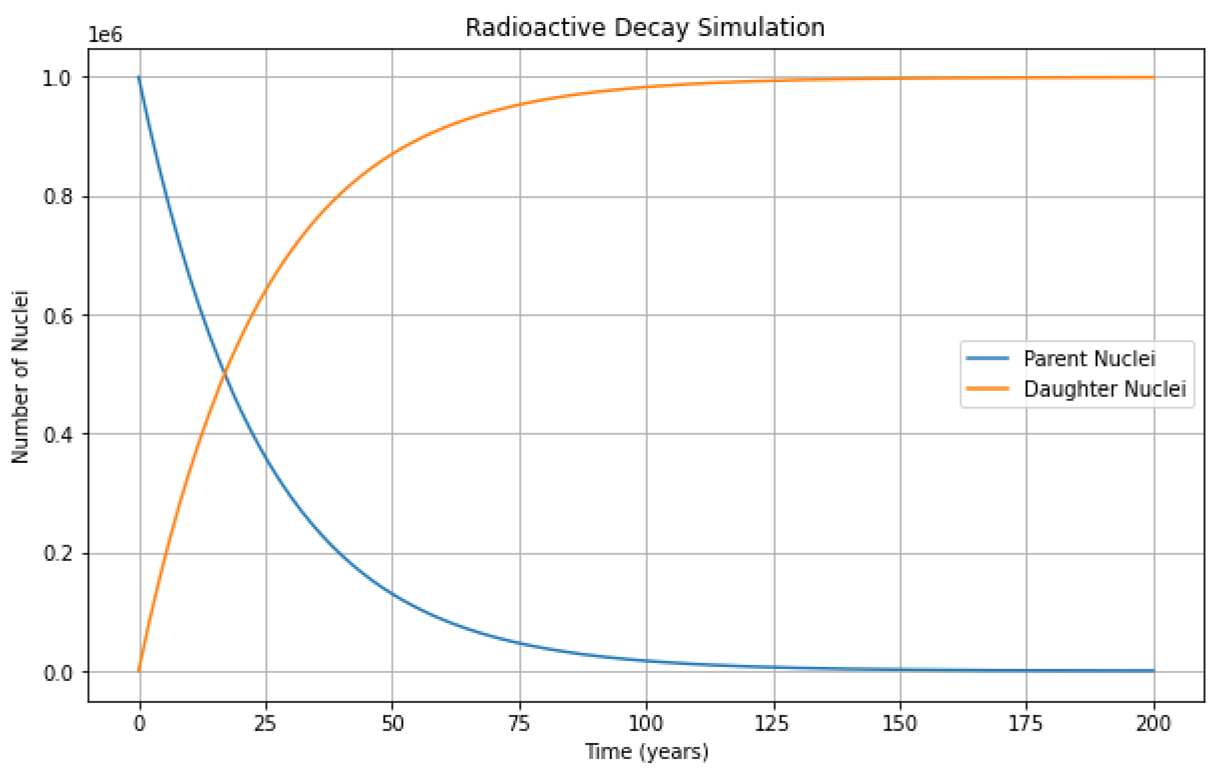
In [1]:

```
# task 1
# N0 at t = 0
# random number testing whether an atam has decayed in a time step

# Import the python libraries needed to carry out the caluclations and graphs
import numpy as np
import matplotlib.pyplot as plt

# instruction to the kernel to prepare for plot instructions
%matplotlib inline

# setting up constants
decay_p1 = 0.04 # probability of decay, 4% chance of decay in 1 second of inital sample
N_0 = 1e6 # initial number of nuclei at time t = 0
D_0 = 0 # initial number of daughter atoms at t = 0
decay_constant1 = np.log(1 / (1 - decay_p1))
t_half1 = (np.log(2)) / decay_constant1

# Determine number of jumps required
t_max = 200 # Iterate up to a maximum of t_max (years)
delta_t = 1 # Set the time step as one year
N = int(t_max / delta_t) # Calculate the number of time jumps

# Set up the NumPy arrays to hold the Number of Nuclei N_nuc and time t values and initialise
# Need one more element in each array than the number of jumps, to include the zeroth element
N_nuc = np.zeros(N + 1)
D_nuc = np.zeros(N + 1)
#t = np.zeros(N + 1)
N_nuc[0] = N_0
D_nuc[0] = D_0 # explicitly stating daughter atoms start at zero
#t[0] = 0 # explicitly stating time zero
t = np.arange(0, t_max + 1, delta_t )

# Use a for loop to step along the t-axis N times
# Use the generalised formula for Euler's technique stated in description
for i in range(1, len(t)): # iterating along time axis
    # Calculate decay with an if statement
    for j in range(int(N_nuc[i-1])):
        if np.random.rand() < decay_p1:
            N_nuc[i] = N_nuc[i-1] * (np.exp(- decay_constant1 * delta_t))  # Estimate N at the end of the interval
            D_nuc[i] = D_nuc[i-1] + (N_nuc[i-1] - N_nuc[i])  # estimate D at the end of the interval

            N_nuc[i] = int(N_nuc[i]) - 1
            D_nuc[i] = int(D_nuc[i]) + 1

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(t, N_nuc, label='Parent Nuclei')
plt.plot(t, D_nuc, label='Daughter Nuclei')
plt.xlabel('Time (years)')
plt.ylabel('Number of Nuclei')
plt.title('Radioactive Decay Simulation')
plt.legend()
plt.grid()
plt.show()

# Estimating the half-life from the simulation
absolute_value = np.abs(N_nuc - N_0/2) # calculating the absolute value of the difference between N_nuc and half the initial number of nuclei
# do this for each time step, using argmin to find the index of the minimum value in the resulting array
# then extract the corresponding time value and print it
half_life_estimate = t[np.argmin(absolute_value)]
print('Estimated half-life: {0:.2f} years'.format(half_life_estimate))

# Comparing with the theoretical expectation
print('Theoretical half-life: {0:.2f} years'.format(t_half1))
```

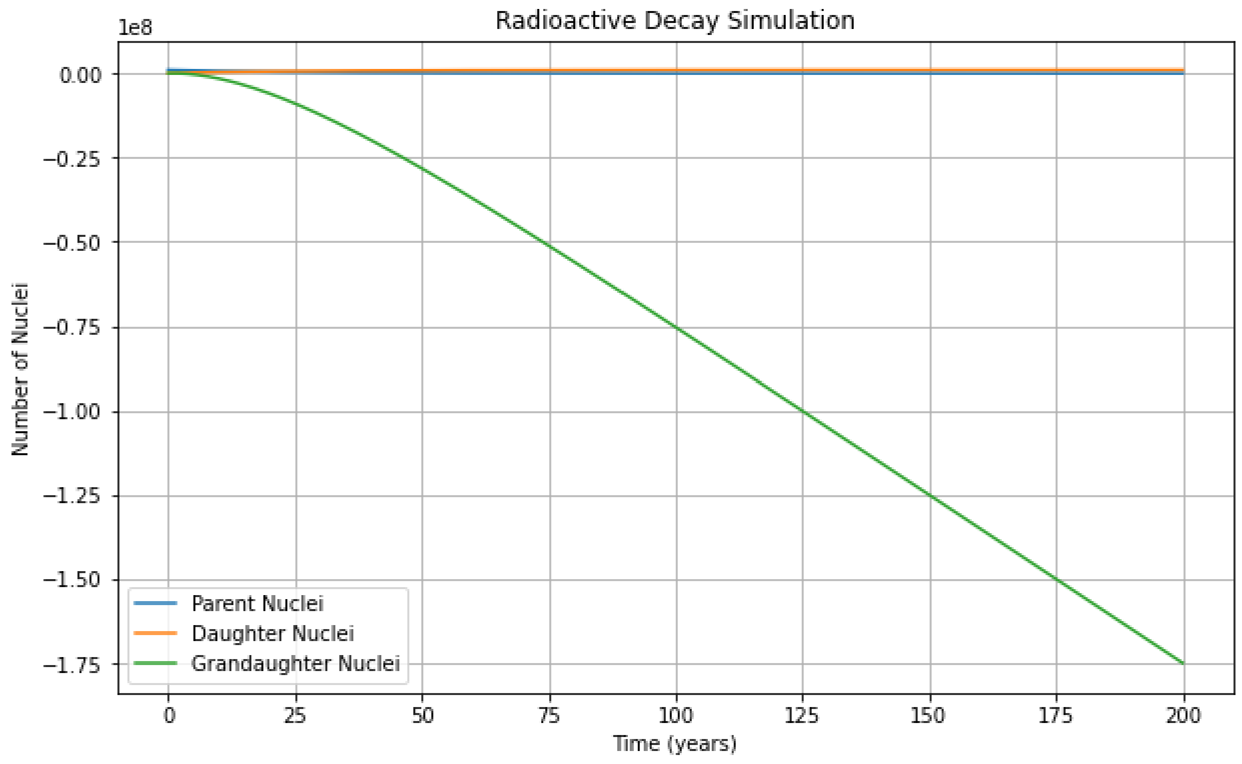



```
Estimated half-life: 17.00 years
Theoretical half-life: 16.98 years
```

In [3]:

```
# task 2

# setting up constants
decay_p2 = 0.015
N_0 = 1e6
decay_constant2 = np.log(1 / (1 - decay_p2))
t_half2 = (np.log(2)) / decay_constant2

# same as before, determine number of steps required
t_maxi = 500 # Iterate up to a maximum of t_max (years)
delta_t = 1 # Set the time step as one year
Num = int(t_maxi/delta_t) # Calculate the number of time jumps

# set up NumPy arrays and initialise again, this time including grandaughter nuclei
# won't do the initial mother nuclei as this equation stays the same
N_nuc = np.zeros(N + 1)
D_nuc1 = np.zeros(N + 1)
G_nuc = np.zeros(N + 1)
N_nuc[0] = N_0
D_nuc1[0] = D_0
G_nuc[0] = 0
t = np.arange(0, t_max + 1, delta_t )


# Use a for loop to step along the t-axis N times
# Use the generalised formula for Euler's technique stated in description
for i in range(1, len(t)):
    # Calculate decay with an if statement
    for j in range(int(N_nuc[i-1])):
        if np.random.rand() < decay_p1:
            N_nuc[i] = N_nuc[i-1] * (np.exp(- decay_constant1 * delta_t))  # Estimate N at the end of the interval
            D_nuc[i] = D_nuc[i-1] + (N_nuc[i-1] - N_nuc[i])  # estimate D at the end of the interval

            N_nuc[i] = int(N_nuc[i]) - 1
            D_nuc[i] = int(D_nuc[i]) + 1

    for k in range(int(D_nuc[i-1])):
        if np.random.rand() < decay_p2:
```

```
            D_nuc1[i] = D_nuc1[i-1] + (((decay_constant1 * N_nuc[i-1]) / (decay_constant1 - decay_constant2)) * (np.exp(-decay_constant2 * delta_t) - np.exp(-decay_constant1 * delta_t)))  # est
            G_nuc[i] = G_nuc[i-1] + (N_nuc[i-1] - N_nuc[i] - D_nuc1[i])

            D_nuc1[i] = int(D_nuc[i]) - 1
            G_nuc[i] = int(G_nuc[i]) + 1

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(t, N_nuc, label='Parent Nuclei')
plt.plot(t, D_nuc, label='Daughter Nuclei')
plt.plot(t, G_nuc, label='Grandaughter Nuclei')
plt.xlabel('Time (years)')
plt.ylabel('Number of Nuclei')
plt.title('Radioactive Decay Simulation')
plt.legend()
plt.grid()
plt.show()

# # Estimating the half-life from the simulation
# absolute_value = np.abs(N_nuc - N_0/2) # calculating the absolute value of the difference between N_nuc and half the initial number of nuclei
# # do this for each time step, using argmin to find the index of the minimum value in the resulting array
# # then extract the corresponding time value and print it
# half_life_estimate = t[np.argmin(absolute_value)]
# print('Estimated half-life: {0:.2f} years'.format(half_life_estimate))

# # Comparing with the theoretical expectation
# print('Theoretical half-life: {0:.2f} years'.format(t_half1))
```



In [ ]: