Numerical Methods :
Rate equations or MC approaches for modelling growth
- Task 0 & Task 1 Report -

Léo BECHET, M2 CompuPhys 2024-2025

Task 0

Examples of physical systems that can be modeled using rate equations.

1. **Radioactive Decay**
   Radioactive materials decay at a rate proportional to the number of undecayed material, which can be modeled by the rate equation: $\frac{dN}{dt} = -\lambda N$ where N is the number of undecayed nuclei and λ is the decay constant.

2. **Chemical Reactions**
   In chemical reactions of type $A => B$, the rate at which A transforms into B is governed by the rate equation: $\frac{d[A]}{dt} = -k[A]$ where $[A]$ is the concentration of A and k is the rate constant.

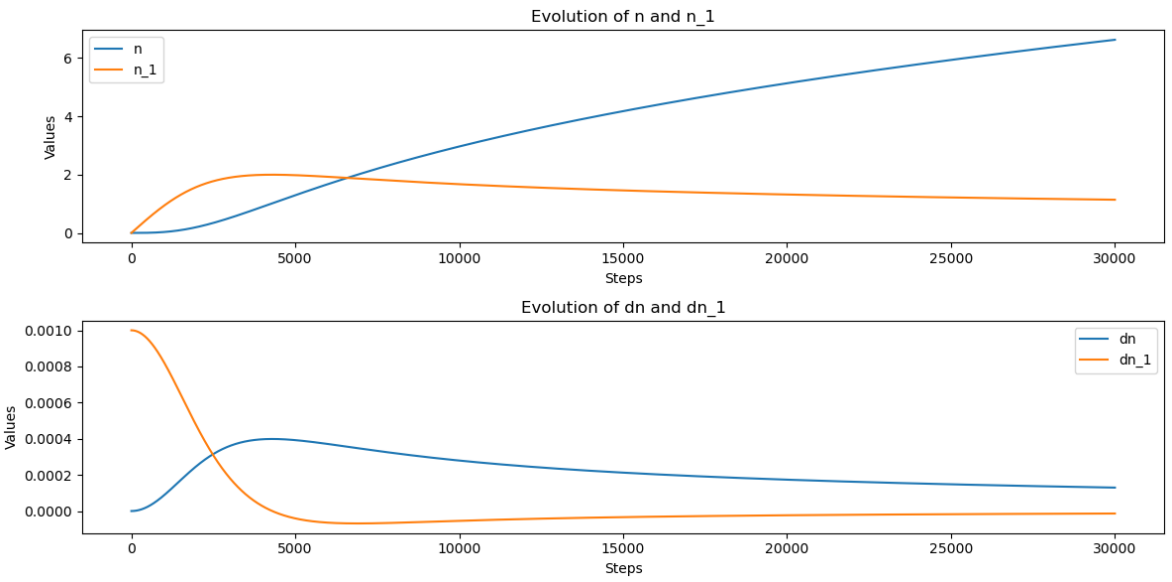
3. **Population Growth**
   A population of organisms growing without resource limitations can be modeled by the logistic growth equation: $\frac{dN}{dt} = rN(1 - \frac{N}{K})$ where N is the population size, r is the growth rate, and K is the carrying capacity.

Task 1

Basic integration

In [1]:
```python
import matplotlib.pyplot as plt

# Sample data for demonstration purposes
steps = 30000
n = 0
n_1 = 0
F_a = 1
F_d = 0
D_1 = 0.1
alpha = 0

dt = 0.001

# Initialize lists
```

```
l_n = [n]
l_n1 = [n_1]
l_dn = []
l_dn1 = []

# Simulate the loop
for i in range(steps):
    dn_1 = (F_a - F_d - 2 * D_1 * n_1**2 - D_1 * n_1 * n - alpha * F_a *
    dn = (D_1 * n_1**2 + alpha * F_a * n_1) * dt

    n += dn
    n_1 += dn_1

    l_n.append(n)
    l_n1.append(n_1)

    l_dn.append(dn)
    l_dn1.append(dn_1)

# Plotting the evolution of n and n_1
plt.figure(figsize=(12, 6))

# First subplot
plt.subplot(2, 1, 1)
plt.plot(l_n, label='n')
plt.plot(l_n1, label='n_1')
plt.title('Evolution of n and n_1')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

# Second subplot
plt.subplot(2, 1, 2)
plt.plot(l_dn, label='dn')
plt.plot(l_dn1, label='dn_1')
plt.title('Evolution of dn and dn_1')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

plt.tight_layout()
plt.show()
```

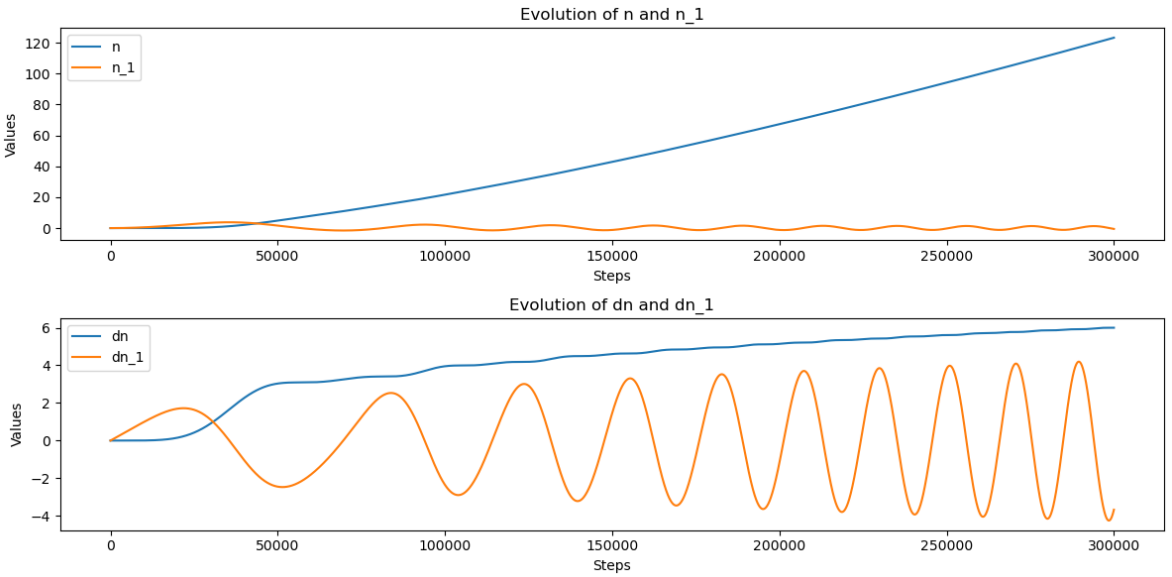


The simple integration scheme using a dt multiplication may be simple and not suitable for long integration, but does offer the benefit of being exact when $dt => 0$. It is always a good idea to try it and see if the results are as expected for different values of dt, as it can give an idea of the characteristic time of evolution of the system (i.e. fast evolving system will require smaller dt, whereas slow ones will allow the use of greater values).

Test implementation of velocity verlet scheme

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the system of differential equations (derivatives)
def compute_derivatives(n, n_1, F_a, F_d, D_1, alpha):
    dn_dt = D_1 * n_1**2 + alpha * F_a * n_1
    dn_1_dt = F_a - F_d - 2 * D_1 * n_1**2 - D_1 * n_1 * n - alpha * F_a
    return dn_dt, dn_1_dt

# Parameters
F_a = 1
F_d = 0
D_1 = 0.1
alpha = 0
dt = 0.0001
steps = 300000

# Initialize arrays
n = np.zeros(steps)
n_1 = np.zeros(steps)
v_n = np.zeros(steps)
v_n_1 = np.zeros(steps)

# Initial conditions
n[0] = 0
n_1[0] = 0
# Assuming initial velocities are zero, adjust if needed
v_n[0] = 0
v_n_1[0] = 0

# Velocity Verlet integration
for i in range(1, steps):
    # Compute accelerations at the previous step
    dn_dt, dn_1_dt = compute_derivatives(n[i-1], n_1[i-1], F_a, F_d, D_1,

    # Update positions
    n[i] = n[i-1] + v_n[i-1] * dt + 0.5 * dn_dt * dt**2
    n_1[i] = n_1[i-1] + v_n_1[i-1] * dt + 0.5 * dn_1_dt * dt**2

    # Compute new accelerations with updated positions
    dn_dt_new, dn_1_dt_new = compute_derivatives(n[i], n_1[i], F_a, F_d,

    # Update velocities
    v_n[i] = v_n[i-1] + 0.5 * (dn_dt + dn_dt_new) * dt
    v_n_1[i] = v_n_1[i-1] + 0.5 * (dn_1_dt + dn_1_dt_new) * dt

# Plotting the results
```

```
plt.figure(figsize=(12, 6))

# First subplot
plt.subplot(2, 1, 1)
plt.plot(n, label='n')
plt.plot(n_1, label='n_1')
plt.title('Evolution of n and n_1')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

# Compute dn and dn_1
dn = np.gradient(n, dt)
dn_1 = np.gradient(n_1, dt)

# Second subplot
plt.subplot(2, 1, 2)
plt.plot(dn, label='dn')
plt.plot(dn_1, label='dn_1')
plt.title('Evolution of dn and dn_1')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

plt.tight_layout()
plt.show()
```

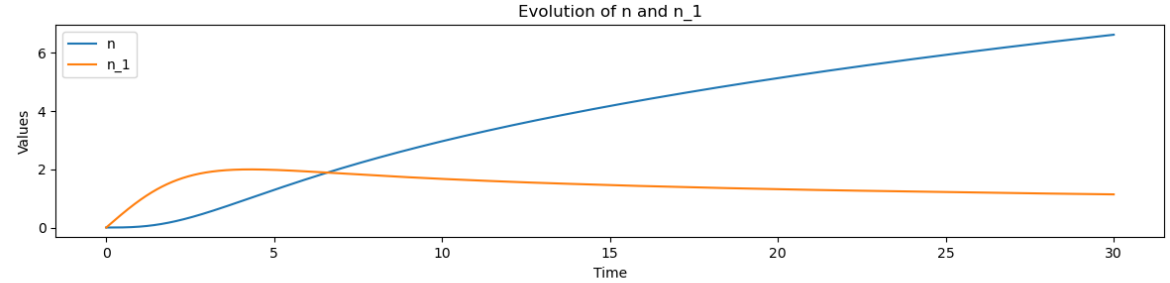


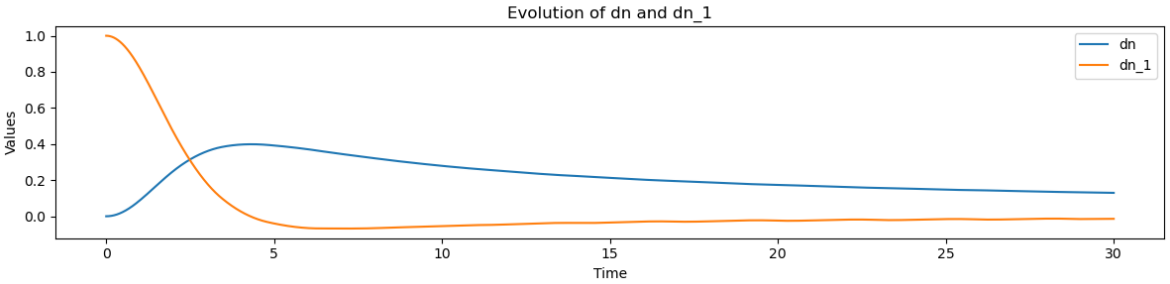
The results out from that velocity verlet integration are wrong. Though we suspect a bad implementation from our part, it seems that it is not adapted to our system. The oscillations have no reason of existing since we are talking about densities and the value cannot go under 0. Furthermore, different values of dt gives the same result, outliying the importance of having an idea of what one should observe following integration.

Runge-Kuta 5(4) integration

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

```python
# Define the system of differential equations
def system(t, y, F_a, F_d, D_1, alpha):
    n, n_1 = y
    dn_1_dt = F_a - F_d - 2 * D_1 * n_1**2 - D_1 * n_1 * n - alpha * F_a
    dn_dt = D_1 * n_1**2 + alpha * F_a * n_1
    return [dn_dt, dn_1_dt]

# Parameters
F_a = 1
F_d = 0
D_1 = 0.1
alpha = 0

# Initial conditions
n0 = 0
n_10 = 0
y0 = [n0, n_10]

# Time span
t_span = (0, 30)  # Adjust the end time as needed
t_eval = np.linspace(t_span[0], t_span[1], 30000)

# Solve the ODEs using RK45
sol = solve_ivp(system, t_span, y0, args=(F_a, F_d, D_1, alpha), method='

# Extract the results
n = sol.y[0]
n_1 = sol.y[1]

# Plotting the results
plt.figure(figsize=(12, 6))

# First subplot
plt.subplot(2, 1, 1)
plt.plot(sol.t, n, label='n')
plt.plot(sol.t, n_1, label='n_1')
plt.title('Evolution of n and n_1')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()

# Compute dn and dn_1
dn = np.gradient(n, sol.t)
dn_1 = np.gradient(n_1, sol.t)

# Second subplot
plt.subplot(2, 1, 2)
plt.plot(sol.t, dn, label='dn')
plt.plot(sol.t, dn_1, label='dn_1')
plt.title('Evolution of dn and dn_1')
plt.xlabel('Time')
plt.ylabel('Values')
plt.legend()

plt.tight_layout()
plt.show()
```





Runge-Kuta 5(4) using scipy

Scipy has a number of prebuilt integrators including RK45 and other ODEs. These are very well optimised algorithms that are suitable for most modern problems, unless special integration schemes are required. RK45 is a scheme well known for its stability and adaptability to different problems, thus, testing it is a logical step.

Final thoughts on integration scheme

While Velocity-Verlet didn't show promising results, the basic dt multiplication integration and RK45 schemes do show the correct evolution of the system. Though most likely an error from our part, the Velocity-Verlet case illustrates the need of one to be understand the system he is looking at to spot odd results.

Here we will use the Basic Integration schemes, as the RK45 case has already been heavily documented.

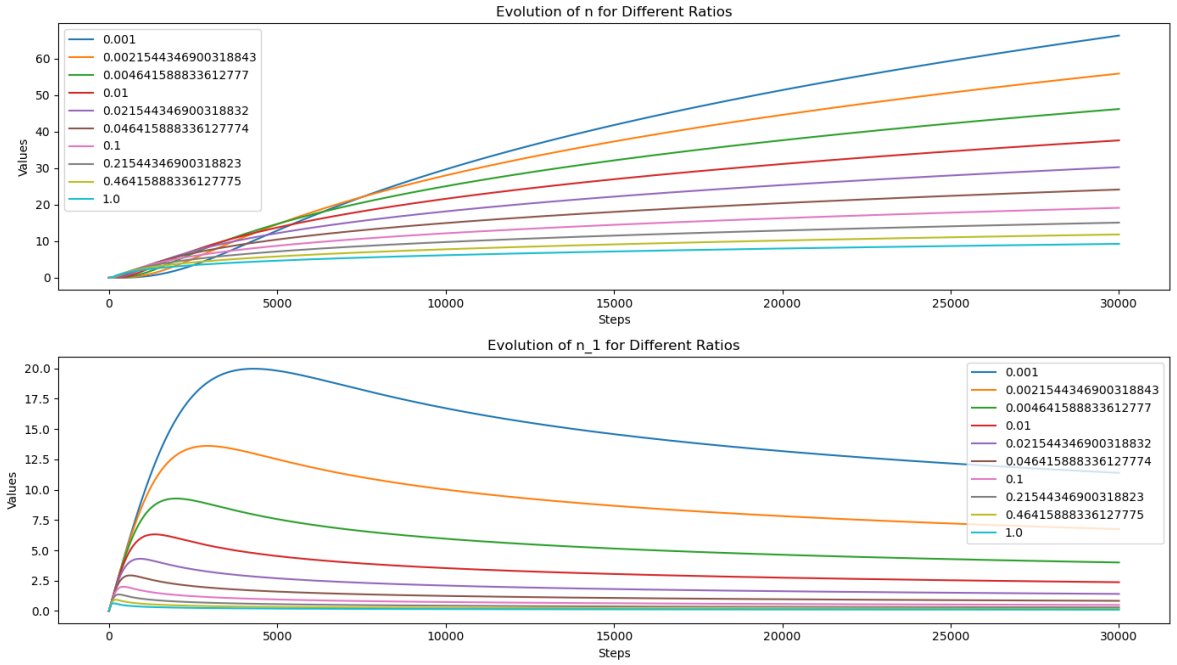
Grid display of final values of n and n_1 in function of $\frac{F_a}{D_1}$ using Basic Integration

Having $F_D = 0$ means we consider our Monomers to be immortal, i.e. they will not die and there are no disparition of Monomers. α represents the interaction rate. Setting it to 0 means that we do not take into consideration the interactions between monomers. Instead, islands are only created due to other factors.

In [4]:
```
# Here we use F_d and alpha = 0
import numpy as np
import matplotlib.pyplot as plt

# Parameters
steps = 30000
F_d = 0
```

```
alpha = 0
dt = 0.01

# List of ratios to test
ratios = np.logspace(-3, 0,10)  # Example ratios, you can change or expan

# Initialize storage for results
results = {ratio: {'n': [], 'n_1': [], 'dn': [], 'dn_1': []} for ratio in

# Simulate for each ratio
for ratio in ratios:
    D_1 = ratio    # D_1 is set to 1 for simplicity
    F_a = 1.0  # F_a is set to the current ratio

    # Initial conditions
    n = 0
    n_1 = 0

    # Initialize lists for this ratio
    l_n = [n]
    l_n1 = [n_1]
    l_dn = []
    l_dn1 = []

    # Simulate the loop
    for i in range(steps):
        dn_1 = (F_a - F_d - 2 * D_1 * n_1**2 - D_1 * n_1 * n - 2 * alpha
        dn = (D_1 * n_1**2 + alpha * F_a * n_1) * dt

        n += dn
        n_1 += dn_1

        l_n.append(n)
        l_n1.append(n_1)
        l_dn.append(dn)
        l_dn1.append(dn_1)

    # Store results
    results[ratio]['n'] = l_n
    results[ratio]['n_1'] = l_n1
    results[ratio]['dn'] = l_dn
    results[ratio]['dn_1'] = l_dn1

# Plotting the results
plt.figure(figsize=(14, 8))

# Plot evolution of n and n_1 for each ratio
plt.subplot(2, 1, 1)
for ratio in ratios:
    plt.plot(results[ratio]['n'], label=f'{ratio}')
plt.title('Evolution of n for Different Ratios')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

# Plot evolution of dn and dn_1 for each ratio
plt.subplot(2, 1, 2)
for ratio in ratios:
    plt.plot(results[ratio]['n_1'], label=f'{ratio}')
plt.title('Evolution of n_1 for Different Ratios')
```

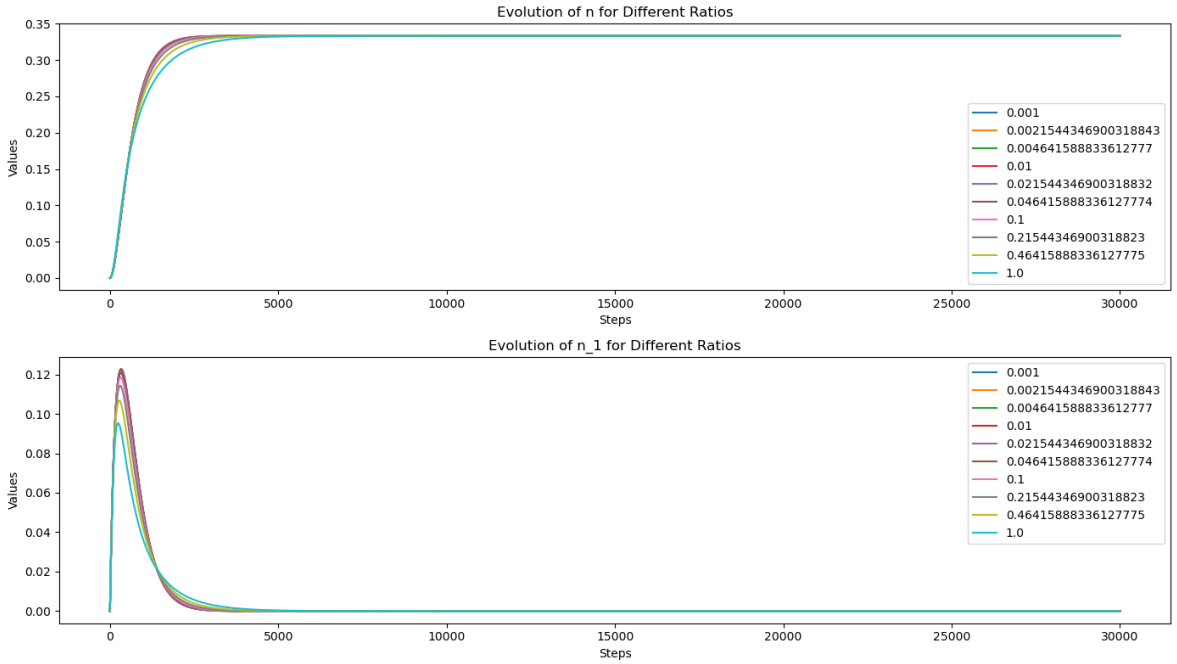
```
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

plt.tight_layout()
plt.show()
```



The smaller the ratio, the bigger the bump in n_1 at the start of the simulation, and the slower the increase for n. We can also see that depending on the ratio, n_1 tends to smaller values, where n seems to keep growing in a logarithmic fashion. One could run longer simulations to deduce the tendency of the curves, but it would always require longer and longer compute time.

In [5]:
```
# Here we use F_d and alpha = 0
import numpy as np
import matplotlib.pyplot as plt

# Parameters
steps = 30000
F_d = 0.9
alpha = 0.3
dt = 0.01

# List of ratios to test
ratios = np.logspace(-3, 0,10)  # Example ratios, you can change or expan

# Initialize storage for results
results = {ratio: {'n': [], 'n_1': [], 'dn': [], 'dn_1': []} for ratio in

# Simulate for each ratio
for ratio in ratios:
    D_1 = ratio    # D_1 is set to 1 for simplicity
    F_a = 1.0  # F_a is set to the current ratio

    # Initial conditions
    n = 0
    n_1 = 0
```

```
    # Initialize lists for this ratio
    l_n = [n]
    l_n1 = [n_1]
    l_dn = []
    l_dn1 = []

    # Simulate the loop
    for i in range(steps):
        dn_1 = (F_a - F_d - 2 * D_1 * n_1**2 - D_1 * n_1 * n - 2 * alpha
        dn = (D_1 * n_1**2 + alpha * F_a * n_1) * dt

        n += dn
        n_1 += dn_1

        l_n.append(n)
        l_n1.append(n_1)
        l_dn.append(dn)
        l_dn1.append(dn_1)

    # Store results
    results[ratio]['n'] = l_n
    results[ratio]['n_1'] = l_n1
    results[ratio]['dn'] = l_dn
    results[ratio]['dn_1'] = l_dn1

# Plotting the results
plt.figure(figsize=(14, 8))

# Plot evolution of n and n_1 for each ratio
plt.subplot(2, 1, 1)
# plt.ylim(0,10)
for ratio in ratios:
    plt.plot(results[ratio]['n'], label=f'{ratio}')
plt.title('Evolution of n for Different Ratios')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

# Plot evolution of dn and dn_1 for each ratio
plt.subplot(2, 1, 2)
# plt.ylim(0,10)
for ratio in ratios:
    plt.plot(results[ratio]['n_1'], label=f'{ratio}')
plt.title('Evolution of n_1 for Different Ratios')
plt.xlabel('Steps')
plt.ylabel('Values')
plt.legend()

plt.tight_layout()
plt.show()
```



We now include interactions between monomers using α. As time goes on, the density of monomers first increases a lot before dropping down. Meanwhile, the density of islands increases and stabilizes. This is due to the fact that at first, there are no islands and monomers encountering eachother has a low probability, leading to a rapid growth. However after more monomers have been added, it becomes quite likely to run into one, thus interactions lead to the creation of islands, in turns increasing the chance of monomers dimerizing with islands and other monomers. Finally, islands become the main part of the space, and new monomers are quickly encountering them and turning into dimers, thus the monomers density tends to 0. Islands growth becomes slow thus stabilizing towards a single value.