Modélisation d'une épidémie dans un campus universitaire

Codjo Ulrich Expéra AKAKPO

29 août 2025

Table des matières

1. Introduction

La propagation des maladies infectieuses est un phénomène complexe qui a depuis longtemps suscité l'intérêt des chercheurs, tant en biologie qu'en mathématiques. L'une des approches les plus influentes dans ce domaine est la modélisation mathématique des épidémies, dont l'objectif est de comprendre, prédire et contrôler la dynamique de transmission des agents pathogènes au sein d'une population.

Parmi les premiers à formaliser ce type de modélisation, on trouve les travaux pionniers de William Ogilvy Kermack et Anderson Gray McKendrick en 1927, qui ont introduit le célèbre modèle SIR (Susceptible-Infectious-Recovered). Ce modèle compartimente la population en trois groupes : les individus susceptibles d'être infectés (S), les individus infectieux (I), et ceux qui sont rétablis ou immunisés (R). Leur cadre mathématique simple mais puissant a jeté les bases de l'épidémiologie théorique moderne.

Depuis, le modèle SIR a été étendu et adapté à de nombreux contextes : maladies à immunité temporaire, modèles SEIR (ajout d'une phase d'incubation), dynamiques spatiales, effets de la vaccination, etc. Il constitue un outil essentiel pour les autorités sanitaires et les chercheurs, permettant par exemple d'estimer des seuils critiques de vaccination, ou de simuler l'impact de mesures de confinement.

Dans ce projet, nous nous appuyons sur une version enrichie du modèle SIR, qui prend en compte la diffusion spatiale de l'épidémie sur un réseau de bâtiments. Cette approche permet de capturer plus finement la réalité d'un campus universitaire, où les interactions entre individus sont fortement structurées par la géographie et l'architecture du lieu.

2. Contextualisation et modélisation

Dans ce projet, nous cherchons à modéliser la propagation d'une épidémie de grippe dans un campus universitaire composé de 20 bâtiments. Chaque bâtiment héberge une population d'individus qui peuvent interagir entre eux localement, mais également se déplacer d'un bâtiment à un autre. Ces déplacements sont responsables de la diffusion spatiale de la maladie à travers le campus.

Pour représenter ce phénomène, nous utilisons une version spatialisée du modèle SIR classique (Susceptibles – Infectés – Rétablis), formulée pour chaque bâtiment du campus.

2.1. Modèle mathématique

Pour chaque bâtiment i, on modélise l'évolution dans le temps des trois sous-populations suivantes :

— $S_i(t)$: nombre d'individus **susceptibles** (non infectés mais vulnérables),

— $I_i(t)$: nombre d'individus **infectés** (porteurs de la maladie et contagieux),

— $R_i(t)$: nombre d'individus **rétablis** (guéris ou immunisés).

Le système d'équations différentielles régissant cette dynamique est le suivant :

$$\frac{dS_i}{dt} = -\beta \frac{S_i I_i}{N_i} + D_S \nabla^2 S_i \tag{1}$$

$$\frac{dI_i}{dt} = \beta \frac{S_i I_i}{N_i} - \gamma I_i + D_I \nabla^2 I_i \tag{2}$$

$$\frac{dR_i}{dt} = \gamma I_i + D_R \nabla^2 R_i \tag{3}$$

où $N_i = S_i + I_i + R_i$ est la population totale du bâtiment i, et ∇^2 est l'opérateur de Laplace discret modélisant la diffusion entre bâtiments voisins.

2.2. Interprétation des équations

Équation (1) – Susceptibles :

- $\beta \frac{S_i I_i}{N_i}$: modélise la contamination des individus sains.
  - β est le **taux de transmission**, représentant la probabilité de contagion lors d'un contact.
  - $\frac{S_i I_i}{N_i}$ approxime le nombre de contacts susceptibles-infectés.
  - Ce terme diminue S_i.
- $+D_S \nabla^2 S_i$: représente la diffusion spatiale des susceptibles.
  - D_S est le coefficient de diffusion.
  - Ce terme modélise les mouvements de personnes saines entre bâtiments.

Équation (2) – Infectés :

- $\beta \frac{S_i I_i}{N_i}$: représente les nouvelles infections (même mécanisme que dans (1)).
- $-\gamma I_i$: modélise la guérison des personnes infectées.
  - γ est le **taux de guérison**.
  - Ce terme diminue I_i.
- $+D_I \nabla^2 I_i$: diffusion spatiale des individus infectés entre bâtiments.

Équation (3) – Rétablis :

- γI_i : individus infectés qui guérissent.
- $+D_R \nabla^2 R_i$: diffusion spatiale des rétablis (individus guéris ou immunisés).

Au total, ce système constitue un **modèle SIR spatialement distribué**, adapté à un environnement structuré (tel qu'un campus). Il combine deux dynamiques :

- Une dynamique épidémique **locale** (au sein de chaque bâtiment),
- Une dynamique **spatiale** (déplacements entre bâtiments).

3. Résolution numérique

Le système présenté est **non linéaire** et **couplé spatialement**, ce qui rend toute résolution analytique impossible pour un grand nombre de bâtiments. Pour obtenir une approximation de l'évolution temporelle de l'épidémie, nous recourons à des **méthodes numériques de discrétisation**.

Plus précisément, nous appliquons trois méthodes classiques :

— **La méthode d'Euler explicite** : rapide et simple à implémenter, mais sujette à des instabilités numériques pour certains pas de temps. Elle consiste à approximer la dérivée temporelle en évaluant le système à l'instant courant uniquement.

— **La méthode d'Euler implicite** : plus stable, nécessitant toutefois la *résolution d'un système non linéaire* à chaque pas de temps. Pour cela, nous utilisons la **méthode de Newton-Raphson**, permettant de linéariser et résoudre efficacement les équations implicites.

— **La méthode de Runge-Kutta d'ordre 4 (RK4)** : plus précise que les méthodes d'Euler, elle réalise plusieurs évaluations intermédiaires du système à chaque pas de temps. Elle offre un excellent compromis entre précision et complexité algorithmique, tout en restant explicite.

3.1. Méthode d'Euler explicite

3.1.1. Discrétisation temporelle et spatiale

a- Discrétisation temporelle

Les équations suivantes :

$$\frac{dS_i}{dt} = f_S(S_i, I_i, R_i)$$
$$\frac{dI_i}{dt} = f_I(S_i, I_i, R_i)$$
$$\frac{dR_i}{dt} = f_R(S_i, I_i, R_i)$$

seront approchées numériquement à chaque pas de temps Δt.

On approxime grâce aux différences centrées avant et on obtient :

$$\frac{dS_i}{dt} \approx \frac{S_i^{n+1} - S_i^n}{\Delta t}$$

Ce qui donne :

$$S_i^{n+1} = S_i^n + \Delta t \left(-\beta \frac{S_i^n I_i^n}{N_i^n} + D_S \nabla^2 S_i^n \right)$$
$$I_i^{n+1} = I_i^n + \Delta t \left(\beta \frac{S_i^n I_i^n}{N_i^n} - \gamma I_i^n + D_I \nabla^2 I_i^n \right)$$
$$R_i^{n+1} = R_i^n + \Delta t \left(\gamma I_i^n + D_R \nabla^2 R_i^n \right)$$

Avec :

— S_i^n : nombre de susceptibles dans le bâtiment i au temps $t = n\Delta t$

— Idem pour I_i^n, R_i^n

— S_i^{n+1} : valeur au temps suivant

— $\nabla^2 S_i$: approximé par une discrétisation spatiale (on verra dans la section suivante)

b- Discrétisation spatiale

Les équations de SIR qu'on a vues comprennent des termes de diffusion spatiale :

$$\nabla^2 S_i^n \quad , \nabla^2 I_i^n \quad , \nabla^2 R_i^n$$

Mais ici on ne travaille pas dans l'espace continu (comme sur une carte), mais dans un réseau de bâtiments connectés, comme un graphe. Soit

$$X_i^n = \{S_i^n,\ I_i^n,\ R_i^n\}$$

Il faut donc approximer $\nabla^2 X_i^n$ par une formule discrète adaptée à ce réseau.

* Méthode des Différences Finies

On utilise une formule réseau du Laplacien :

$$\nabla^2 X_i^n \approx \sum_{j \in \mathcal{V}(i)} \frac{X_j^n - X_i^n}{\Delta x^2}$$

Avec :

- $\mathcal{V}(i)$: voisins du bâtiment i
- Δx : distance entre les bâtiments (souvent normalisée à 1)

Cette formule indique que le gradient du bâtiment i dépend de la différence entre ses valeurs et celles de ses voisins.

Exemple : Si i a deux voisins j et k, alors :

$$\nabla^2 X_i \approx \frac{X_j - X_i}{\Delta x^2} + \frac{X_k - X_i}{\Delta x^2}$$

* Méthode des Volumes finis

La méthode des volumes finis repose sur un **principe de conservation** : la variation de la quantité I dans un volume de contrôle V_i est due au **flux net entrant** depuis ses voisins $j \in \mathcal{V}(i)$. Cela donne :

$$\frac{d}{dt} \int_{V_i} I \, dV = \sum_{j \in \mathcal{V}(i)} D \frac{I_j - I_i}{\Delta x} A_{ij}$$

où D est le coefficient de diffusion, A_{ij} l'aire d'interface entre i et j, et Δx leur distance. En divisant par le volume $|V_i|$, on obtient l'évolution ponctuelle de I_i :

$$\frac{dI_i}{dt} = \frac{1}{|V_i|} \sum_{j \in \mathcal{V}(i)} D \frac{I_j - I_i}{\Delta x} A_{ij}$$

Lien direct avec les différences finies

Si on choisit :

$$|V_i| = \Delta x, \quad A_{ij} = 1, \quad D = 1,$$

alors l'expression devient :

$$\frac{dI_i}{dt} = \sum_{j \in \mathcal{V}(i)} \frac{I_j - I_i}{\Delta x^2}$$

Ce qui correspond exactement à la **formule des différences finies centrées** du Laplacien :

$$\nabla^2 I_i \approx \sum_{j \in \mathcal{V}(i)} \frac{I_j - I_i}{\Delta x^2}$$

Ainsi, la méthode des volumes finis, fondée sur la conservation des flux, **généralise et justifie rigoureusement** la méthode des différences finies. Sur un graphe, cette interprétation permet de dériver naturellement le Laplacien discret :

$$\nabla^2 I_i \approx \sum_{j \in \mathcal{V}(i)} \frac{I_j - I_i}{\Delta x^2}$$

utilisé pour modéliser la diffusion inter-bâtiment dans notre système SIR.

Cette équivalence valide mathématiquement notre approche numérique, en garantissant à la fois cohérence physique (volumes finis) et simplicité d'implémentation (différences finies).

Système d'équations final après discrétisation temporelle et spatiale

On implémente donc finalement la méthode numérique :

$$S_i^{n+1} = S_i^n + \Delta t \left(-\beta \frac{S_i I_i}{N_i} + D_S \sum_{j \in V(i)} \frac{S_j - S_i}{\Delta x^2} \right)$$

$$I_i^{n+1} = I_i^n + \Delta t \left(\beta \frac{S_i I_i}{N_i} - \gamma I_i + D_I \sum_{j \in V(i)} \frac{I_j - I_i}{\Delta x^2} \right)$$

$$R_i^{n+1} = R_i^n + \Delta t \left(\gamma I_i + D_R \sum_{j \in V(i)} \frac{R_j - R_i}{\Delta x^2} \right)$$

Algorithme d'implémentation

Algorithm 1 : Simulation SIR avec diffusion spatiale (Euler explicite)

Initialiser les populations S_i, I_i, R_i pour chaque bâtiment i et ses voisins $\mathcal{V}_i$

Définir les paramètres constants : β (taux de transmission), γ (taux de guérison), D_S, D_I, D_R (coefficients de diffusion), Δt (pas de temps), Δx (distance entre les bâtiments)

Définir le temps total T et calculer le nombre d'itérations $N = \frac{T}{\Delta t}$

for chaque pas de temps t allant de 0 à T avec un pas Δt **do**

  for chaque bâtiment $i \in \{1, ..., 20\}$ **do**

    $N_i = S_i + I_i + R_i$

    Calculer la diffusion :

$$\text{diff}_S = \sum_{j \in \mathcal{V}_i} (S_j - S_i), \quad \text{diff}_I = \sum_{j \in \mathcal{V}_i} (I_j - I_i), \quad \text{diff}_R = \sum_{j \in \mathcal{V}_i} (R_j - R_i)$$

    Calculer les variations par Euler explicite :

$$\Delta S_i = -\beta \frac{S_i I_i}{N_i} + D_S \cdot \text{diff}_S$$
$$\Delta I_i = \beta \frac{S_i I_i}{N_i} - \gamma I_i + D_I \cdot \text{diff}_I$$
$$\Delta R_i = \gamma I_i + D_R \cdot \text{diff}_R$$

    Mettre à jour les états :

$$S_i \leftarrow S_i + \Delta t \cdot \Delta S_i, \quad I_i \leftarrow I_i + \Delta t \cdot \Delta I_i, \quad R_i \leftarrow R_i + \Delta t \cdot \Delta R_i$$

  end for

  Enregistrer les nouveaux états dans un historique pour visualisation

end for

3.1.2. Implémentation en Python

Le code suivant implémente la simulation :

```
import matplotlib.pyplot as plt
import ipywidgets as widgets
from ipywidgets import interact
import copy

# Initialisation des batiments
buildings = {
    1:  {"S": 100, "I": 1,  "R": 0, "neighbors": [2, 3]},
    2:  {"S": 120, "I": 0,  "R": 0, "neighbors": [1, 4]},
    3:  {"S": 80,  "I": 0,  "R": 0, "neighbors": [1, 5]},
    4:  {"S": 150, "I": 0,  "R": 0, "neighbors": [2, 6]},
    5:  {"S": 90,  "I": 0,  "R": 0, "neighbors": [3, 7]},
    6:  {"S": 110, "I": 0,  "R": 0, "neighbors": [4, 8]},
    7:  {"S": 95,  "I": 0,  "R": 0, "neighbors": [5, 9]},
    8:  {"S": 130, "I": 0,  "R": 0, "neighbors": [6, 10]},
    9:  {"S": 85,  "I": 0,  "R": 0, "neighbors": [7, 11]},
    10: {"S": 100, "I": 0,  "R": 0, "neighbors": [8, 12]},
    11: {"S": 75,  "I": 0,  "R": 0, "neighbors": [9, 13]},
    12: {"S": 140, "I": 0,  "R": 0, "neighbors": [10, 14]},
    13: {"S": 105, "I": 0,  "R": 0, "neighbors": [11, 15]},
    14: {"S": 115, "I": 0,  "R": 0, "neighbors": [12, 16]},
    15: {"S": 125, "I": 0,  "R": 0, "neighbors": [13, 17]},
    16: {"S": 135, "I": 0,  "R": 0, "neighbors": [14, 18]},
    17: {"S": 145, "I": 0,  "R": 0, "neighbors": [15, 19]},
    18: {"S": 155, "I": 0,  "R": 0, "neighbors": [16, 20]},
    19: {"S": 165, "I": 0,  "R": 0, "neighbors": [17]},
    20: {"S": 175, "I": 0,  "R": 0, "neighbors": [18]},
}

# Parametres du modele
beta = 0.3      # Taux de transmission
gamma = 0.1     # Taux de guerison

D_S = 0.01      # Diffusion des S
D_I = 0.01      # Diffusion des I
D_R = 0.01      # Diffusion des R

dt = 0.1        # Pas de temps
dx = 1.0        # Pas spatial (non utilise ici mais pour reference)
T = 50          # Duree totale
steps = int(T / dt)

# Implementation avec Euler explicite
# On suppose que "history" est une liste de snapshots de l'etat des batiments a chaque iteration
# Chaque element de history est un dict : {1: {"S": ..., "I": ..., "R": ...}, 2: {...}, ..., 20: {...}}
```

```

history = []

for t in range(steps):
    new_buildings = copy.deepcopy(buildings)

    for i in buildings:
        S = buildings[i]["S"]
        I = buildings[i]["I"]
        R = buildings[i]["R"]
        N = S + I + R

        # Terme de diffusion
        diffusion_S = sum((buildings[j]["S"] - S)/dx**2 for j in
           buildings[i]["neighbors"])
        diffusion_I = sum((buildings[j]["I"] - I)/dx**2  for j
           in buildings[i]["neighbors"])
        diffusion_R = sum((buildings[j]["R"] - R)/dx**2  for j
           in buildings[i]["neighbors"])

        # Mises a jour selon Euler explicite
        dS = -beta * S * I / N + D_S * diffusion_S
        dI = beta * S * I / N - gamma * I + D_I * diffusion_I
        dR = gamma * I + D_R * diffusion_R

        new_buildings[i]["S"] += dt * dS
        new_buildings[i]["I"] += dt * dI
        new_buildings[i]["R"] += dt * dR

    buildings = new_buildings
    history.append(copy.deepcopy(buildings))

# Visualisation
def plot_SIR(building_id):
    S_vals = [snapshot[building_id]["S"] for snapshot in history]
    I_vals = [snapshot[building_id]["I"] for snapshot in history]
    R_vals = [snapshot[building_id]["R"] for snapshot in history]

    plt.figure(figsize=(10, 5))
    plt.plot(S_vals, label="S (Susceptibles)", color='blue')
    plt.plot(I_vals, label="I (Infect s)", color='red')
    plt.plot(R_vals, label="R (R tablis)", color='green')
    plt.title(f"  volution   SIR - B timent {building_id}")
    plt.xlabel("Temps (it rations)")
    plt.ylabel("Nombre   d individus  ")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

#  Widget interactif : selection du batiment (entre 1 et 20)
```

```python
interact(plot_SIR, building_id=widgets.IntSlider(min=1, max=20,
    step=1, value=1, description="Batiment ID"))
```

Exemple

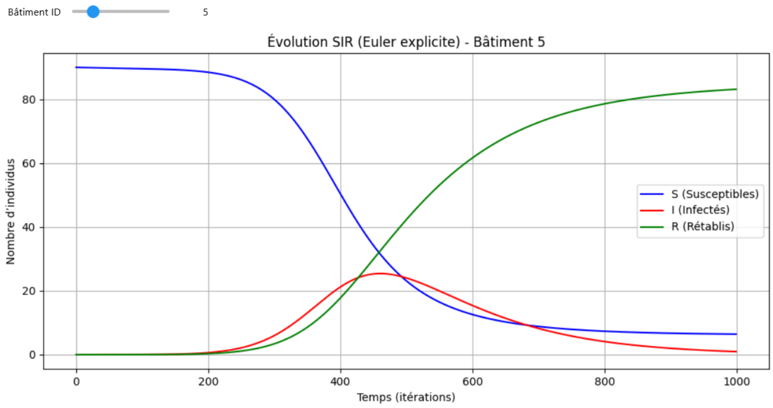

FIGURE 1 – Évolution SIR/ Euler explicite pour le bâtiment 5.

3.2. Méthode d'Euler implicite

3.2.1. Discrétisation temporelle et spatiale

À la différence de l'Euler explicite, ici les dérivées sont approchées en utilisant les valeurs au temps $n+1$:

$$\frac{dS_i}{dt} \approx \frac{S_i^{n+1} - S_i^n}{\Delta t} \quad \Rightarrow \quad S_i^{n+1} = S_i^n + \Delta t \cdot f_S(S_i^{n+1}, I_i^{n+1}, R_i^{n+1})$$

De même pour I_i et R_i :

$$I_i^{n+1} = I_i^n + \Delta t \cdot f_I(S_i^{n+1}, I_i^{n+1}, R_i^{n+1})$$
$$R_i^{n+1} = R_i^n + \Delta t \cdot f_R(S_i^{n+1}, I_i^{n+1}, R_i^{n+1})$$

Mise à jour du système d'équation

Les équations deviennent alors :

$$S_i^{n+1} = S_i^n + \Delta t \left(-\beta \frac{S_i^{n+1} I_i^{n+1}}{N_i^{n+1}} + D_S \sum_{j \in \mathcal{V}(i)} \frac{S_j^{n+1} - S_i^{n+1}}{\Delta x^2} \right)$$

$$I_i^{n+1} = I_i^n + \Delta t \left(\beta \frac{S_i^{n+1} I_i^{n+1}}{N_i^{n+1}} - \gamma I_i^{n+1} + D_I \sum_{j \in \mathcal{V}(i)} \frac{I_j^{n+1} - I_i^{n+1}}{\Delta x^2} \right)$$

$$R_i^{n+1} = R_i^n + \Delta t \left(\gamma I_i^{n+1} + D_R \sum_{j \in \mathcal{V}(i)} \frac{R_j^{n+1} - R_i^{n+1}}{\Delta x^2} \right)$$

Algorithme d'implémentation

Algorithm 2 Simulation SIR avec diffusion (Euler implicite)

Initialiser S_i, I_i, R_i pour chaque bâtiment i
Fixer les paramètres $\beta, \gamma, D_S, D_I, D_R, \Delta t$
for chaque temps $t = 0, \Delta t, 2\Delta t, \ldots$ **do**
    Initialiser les valeurs $S_i^{n+1}, I_i^{n+1}, R_i^{n+1}$ par celles de n
    repeat
        for chaque bâtiment i **do**
            Calculer $N_i^{n+1} = S_i^{n+1} + I_i^{n+1} + R_i^{n+1}$
            Calculer les nouveaux $S_i^{n+1}, I_i^{n+1}, R_i^{n+1}$ selon les équations implicites
        end for
    until convergence (variation inférieure à un seuil ϵ)
    Enregistrer l'état
end for

3.2.2. Implémentation en Python

Le code suivant implémente la simulation :

```python
import matplotlib.pyplot as plt
import ipywidgets as widgets
from ipywidgets import interact
import copy

# Param tres
beta = 0.3
gamma = 0.1
D_S = 0.01
D_I = 0.01
D_R = 0.01
dt = 0.1
dx = 1.0
T = 50
steps = int(T / dt)
dx2 = 1.0

# Initialisation
initial_buildings = {
    1:  {"S": 100, "I": 1,  "R": 0, "neighbors": [2, 3]},
    2:  {"S": 120, "I": 0,  "R": 0, "neighbors": [1, 4]},
    3:  {"S": 80,  "I": 0,  "R": 0, "neighbors": [1, 5]},
    4:  {"S": 150, "I": 0,  "R": 0, "neighbors": [2, 6]},
    5:  {"S": 90,  "I": 0,  "R": 0, "neighbors": [3, 7]},
    6:  {"S": 110, "I": 0,  "R": 0, "neighbors": [4, 8]},
    7:  {"S": 95,  "I": 0,  "R": 0, "neighbors": [5, 9]},
    8:  {"S": 130, "I": 0,  "R": 0, "neighbors": [6, 10]},
    9:  {"S": 85,  "I": 0,  "R": 0, "neighbors": [7, 11]},
    10: {"S": 100, "I": 0,  "R": 0, "neighbors": [8, 12]},
    11: {"S": 75,  "I": 0,  "R": 0, "neighbors": [9, 13]},
    12: {"S": 140, "I": 0,  "R": 0, "neighbors": [10, 14]},
    13: {"S": 105, "I": 0,  "R": 0, "neighbors": [11, 15]},
    14: {"S": 115, "I": 0,  "R": 0, "neighbors": [12, 16]},
```

```
    15: {"S": 125, "I": 0,  "R": 0, "neighbors": [13, 17]},
    16: {"S": 135, "I": 0,  "R": 0, "neighbors": [14, 18]},
    17: {"S": 145, "I": 0,  "R": 0, "neighbors": [15, 19]},
    18: {"S": 155, "I": 0,  "R": 0, "neighbors": [16, 20]},
    19: {"S": 165, "I": 0,  "R": 0, "neighbors": [17]},
    20: {"S": 175, "I": 0,  "R": 0, "neighbors": [18]},
}

def simulate_euler_implicite(buildings_init, steps):
    # Copie de l' tat  initial des b timents pour ne pas modifier l'entr e originale
    buildings = copy.deepcopy(buildings_init)

    # Historique de l' volution  SIR   chaque pas de temps
    history = []

    # Boucle principale de temps (de 0   T avec un pas de dt)
    for _ in range(steps):
        # Initialisation des nouvelles valeurs de S, I, R pour tous les b timents
        S_new = {i: buildings[i]["S"] for i in buildings}
        I_new = {i: buildings[i]["I"] for i in buildings}
        R_new = {i: buildings[i]["R"] for i in buildings}

        # M thode d'Euler implicite : on applique une r solution it rative par point fixe
        for _ in range(20):  # Nombre  d itrations  de point fixe (convergence approximative)
            # Sauvegarde des valeurs pr c dentes ( l itration  k-1)
            S_prev = S_new.copy()
            I_prev = I_new.copy()
            R_prev = R_new.copy()

            # Mise   jour de chaque b timent individuellement
            for i in buildings:
                neighbors = buildings[i]["neighbors"]  # Voisins spatiaux
                Ni = S_prev[i] + I_prev[i] + R_prev[i]  # Population totale du b timent i

                # Terme de diffusion : somme des diff rences avec les voisins
                diff_S = sum((S_prev[j] - S_prev[i])/dx**2  for j in neighbors)
                diff_I = sum((I_prev[j] - I_prev[i])/dx**2  for j in neighbors)
                diff_R = sum((R_prev[j] - R_prev[i])/dx**2  for j in neighbors)

                # Mise   jour des variables S, I, R selon le
```

```
                    sch ma implicite
                # S^{n+1}_i = S^n_i + dt    [ -    S^{n+1} I^{n+1} / N + D    (diffusion) ]
                S_new[i] = buildings[i]["S"] + dt * (
                    -beta * S_prev[i] * I_prev[i] / Ni + D_S * diff_S / dx2
                )
                I_new[i] = buildings[i]["I"] + dt * (
                    beta * S_prev[i] * I_prev[i] / Ni - gamma * I_prev[i] + D_I * diff_I / dx2
                )
                R_new[i] = buildings[i]["R"] + dt * (
                    gamma * I_prev[i] + D_R * diff_R / dx2
                )

        # Mise    jour finale des valeurs des b timents avec les valeurs implicites converg es
        for i in buildings:
            buildings[i]["S"] = S_new[i]
            buildings[i]["I"] = I_new[i]
            buildings[i]["R"] = R_new[i]

        # Enregistrement de   l  tat   du syst me    cet instant dans   l  historique
        history.append(copy.deepcopy(buildings))

    # Retourne l'  volution   compl te sur tout l'intervalle de temps
    return history


# Simulation
history_implicit = simulate_euler_implicite(initial_buildings, steps)

# Affichage interactif
def plot_SIR_implicit(building_id):
    S_vals = [snapshot[building_id]["S"] for snapshot in history_implicit]
    I_vals = [snapshot[building_id]["I"] for snapshot in history_implicit]
    R_vals = [snapshot[building_id]["R"] for snapshot in history_implicit]

    plt.figure(figsize=(10, 5))
    plt.plot(S_vals, label="S (Susceptibles)", color='blue')
    plt.plot(I_vals, label="I (Infect s)", color='red')
    plt.plot(R_vals, label="R (R tablis)", color='green')
    plt.title(f"  volution   SIR (Euler implicite) - B timent {building_id}")
    plt.xlabel("Temps (it rations)")
```

```
    plt.ylabel("Nombre  d individus  ")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

interact(plot_SIR_implicit, building_id=widgets.IntSlider(min=1,
   max=20, step=1, value=1, description="B timent"));
```

Exemple :

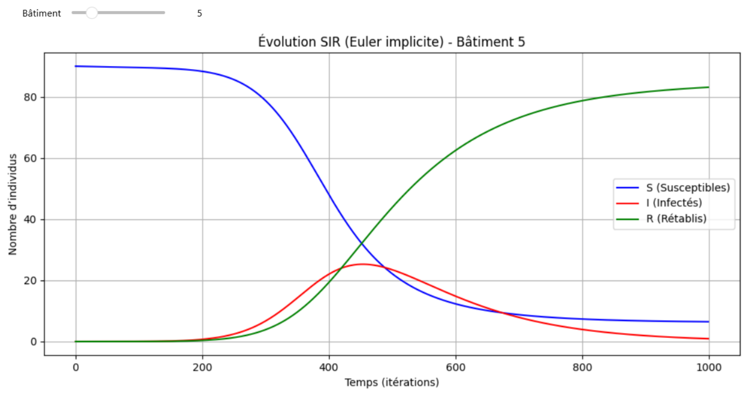

Figure 2 – Évolution SIR/ Euler implicite pour le bâtiment 5.

3.3. Méthode de Runge-Kutta d'ordre 4

Modèle Mathématique

Pour chaque bâtiment i :

$$\frac{dS_i}{dt} = -\beta \frac{S_i I_i}{N_i} + D_S \nabla^2 S_i, \tag{4}$$

$$\frac{dI_i}{dt} = \beta \frac{S_i I_i}{N_i} - \gamma I_i + D_I \nabla^2 I_i, \tag{5}$$

$$\frac{dR_i}{dt} = \gamma I_i + D_R \nabla^2 R_i, \tag{6}$$

où ∇^2 modélise la diffusion entre bâtiments voisins.

3.3.1. Discrétisation temporelle et spatiale

Différences Finies

Laplacien discretisé :

$$\nabla^2 I_i \approx \sum_{j \in \mathcal{V}_i} \frac{I_j - I_i}{\Delta x^2}.$$

Algorithme d'implémentation

Algorithm 3 Simulation SIR avec Méthode Runge-Kutta d'ordre 4

Initialiser les populations S_i, I_i, R_i pour chaque bâtiment i et leurs voisins $\mathcal{V}_i$

Définir les paramètres constants : β (taux d'infection), γ (taux de guérison), D_S, D_I, D_R (coefficients de diffusion), Δt (pas de temps), Δx (distance spatiale)

Définir le temps initial t_0 et le temps total t_{total}

Calculer le nombre d'itérations $n = \frac{t_{\text{total}} - t_0}{\Delta t}$

for each time step t from t_0 to t_{total} by Δt **do**

  Sauvegarder les états actuels S_i^0, I_i^0, R_i^0 pour tous les bâtiments i

  Calculer les dérivées k_{1S}, k_{1I}, k_{1R} avec S_i^0, I_i^0, R_i^0 :

$$k_{1S_i} = -\beta \frac{S_i^0 I_i^0}{S_i^0 + I_i^0 + R_i^0} + D_S \sum_{j \in \mathcal{V}_i} \frac{S_j^0 - S_i^0}{\Delta x^2}$$

$$k_{1I_i} = \beta \frac{S_i^0 I_i^0}{S_i^0 + I_i^0 + R_i^0} - \gamma I_i^0 + D_I \sum_{j \in \mathcal{V}_i} \frac{I_j^0 - I_i^0}{\Delta x^2}$$

$$k_{1R_i} = \gamma I_i^0 + D_R \sum_{j \in \mathcal{V}_i} \frac{R_j^0 - R_i^0}{\Delta x^2}$$

  Calculer les états intermédiaires S_i^1, I_i^1, R_i^1 :

$$S_i^1 = S_i^0 + \frac{\Delta t}{2} k_{1S_i}, \quad I_i^1 = I_i^0 + \frac{\Delta t}{2} k_{1I_i}, \quad R_i^1 = R_i^0 + \frac{\Delta t}{2} k_{1R_i}$$

  Calculer les dérivées k_{2S}, k_{2I}, k_{2R} avec S_i^1, I_i^1, R_i^1

  Calculer les états intermédiaires S_i^2, I_i^2, R_i^2 :

$$S_i^2 = S_i^0 + \frac{\Delta t}{2} k_{2S_i}, \quad I_i^2 = I_i^0 + \frac{\Delta t}{2} k_{2I_i}, \quad R_i^2 = R_i^0 + \frac{\Delta t}{2} k_{2R_i}$$

  Calculer les dérivées k_{3S}, k_{3I}, k_{3R} avec S_i^2, I_i^2, R_i^2

  Calculer les états intermédiaires S_i^3, I_i^3, R_i^3 :

$$S_i^3 = S_i^0 + \Delta t k_{3S_i}, \quad I_i^3 = I_i^0 + \Delta t k_{3I_i}, \quad R_i^3 = R_i^0 + \Delta t k_{3R_i}$$

  Calculer les dérivées k_{4S}, k_{4I}, k_{4R} avec S_i^3, I_i^3, R_i^3

  Mettre à jour les populations :

$$S_i = S_i^0 + \frac{\Delta t}{6}(k_{1S_i} + 2k_{2S_i} + 2k_{3S_i} + k_{4S_i})$$

$$I_i = I_i^0 + \frac{\Delta t}{6}(k_{1I_i} + 2k_{2I_i} + 2k_{3I_i} + k_{4I_i})$$

$$R_i = R_i^0 + \frac{\Delta t}{6}(k_{1R_i} + 2k_{2R_i} + 2k_{3R_i} + k_{4R_i})$$

  Enregistrer les états mis à jour pour l'analyse

end for

3.3.2. Implémentation en Python

Le code suivant implémente la simulation :

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns
from ipywidgets import interact, IntSlider
from IPython.display import display
import copy

# Initialisation des parametres
buildings = {
    1:  {"S": 100, "I": 1,  "R": 0, "neighbors": [2, 3]},
    2:  {"S": 120, "I": 0,  "R": 0, "neighbors": [1, 4]},
    3:  {"S": 80,  "I": 0,  "R": 0, "neighbors": [1, 5]},
    4:  {"S": 150, "I": 0,  "R": 0, "neighbors": [2, 6]},
    5:  {"S": 90,  "I": 0,  "R": 0, "neighbors": [3, 7]},
    6:  {"S": 110, "I": 0,  "R": 0, "neighbors": [4, 8]},
    7:  {"S": 95,  "I": 0,  "R": 0, "neighbors": [5, 9]},
    8:  {"S": 130, "I": 0,  "R": 0, "neighbors": [6, 10]},
    9:  {"S": 85,  "I": 0,  "R": 0, "neighbors": [7, 11]},
    10: {"S": 100, "I": 0,  "R": 0, "neighbors": [8, 12]},
    11: {"S": 75,  "I": 0,  "R": 0, "neighbors": [9, 13]},
    12: {"S": 140, "I": 0,  "R": 0, "neighbors": [10, 14]},
    13: {"S": 105, "I": 0,  "R": 0, "neighbors": [11, 15]},
    14: {"S": 115, "I": 0,  "R": 0, "neighbors": [12, 16]},
    15: {"S": 125, "I": 0,  "R": 0, "neighbors": [13, 17]},
    16: {"S": 135, "I": 0,  "R": 0, "neighbors": [14, 18]},
    17: {"S": 145, "I": 0,  "R": 0, "neighbors": [15, 19]},
    18: {"S": 155, "I": 0,  "R": 0, "neighbors": [16, 20]},
    19: {"S": 165, "I": 0,  "R": 0, "neighbors": [17]},
    20: {"S": 175, "I": 0,  "R": 0, "neighbors": [18]}
}

beta = 0.3      # taux d'infection
gamma = 0.1     # taux de gu rison
DS = 0.01       # diffusion S
DI = 0.01       # diffusion I
DR = 0.01       # diffusion R
dx = 1.0        # distance spatiale
temps_o = 0
temps_total = 100.0  # temps total de simulation
iteration = 500 #Nonbre d'iteration
dt = (temps_total - temps_o)/iteration        # pas de temps


#Discr tisation spatiale (EDP)
def laplacian(var, i, buildings):
    neighbors = buildings[i]["neighbors"]
    return sum((var[j] - var[i]) / dx**2 for j in neighbors)
```

```python


# Fonction de la methode de Runge Kutta a une seule itt ration
def rk4_step(buildings, dt):
    def get_states():
        S = {i: buildings[i]["S"] for i in buildings}
        I = {i: buildings[i]["I"] for i in buildings}
        R = {i: buildings[i]["R"] for i in buildings}
        return S, I, R

    S0, I0, R0 = get_states()

    def deriv(S, I, R):
        dS, dI, dR = {}, {}, {}
        for i in buildings:
            N = S[i] + I[i] + R[i]
            dS[i] = -beta * S[i] * I[i] / N + DS * laplacian(S, i, buildings)
            dI[i] = beta * S[i] * I[i] / N - gamma * I[i] + DI * laplacian(I, i, buildings)
            dR[i] = gamma * I[i] + DR * laplacian(R, i, buildings)
        return dS, dI, dR

    # k1
    k1S, k1I, k1R = deriv(S0, I0, R0)

    # k2
    S2 = {i: S0[i] + k1S[i]/2 for i in buildings}
    I2 = {i: I0[i] + k1I[i]/2 for i in buildings}
    R2 = {i: R0[i] + k1R[i]/2 for i in buildings}
    k2S, k2I, k2R = deriv(S2, I2, R2)

    # k3
    S3 = {i: S0[i] + k2S[i]/2 for i in buildings}
    I3 = {i: I0[i] + k2I[i]/2 for i in buildings}
    R3 = {i: R0[i] + k2R[i]/2 for i in buildings}
    k3S, k3I, k3R = deriv(S3, I3, R3)

    # k4
    S4 = {i: S0[i] + k3S[i] for i in buildings}
    I4 = {i: I0[i] + k3I[i] for i in buildings}
    R4 = {i: R0[i] + k3R[i] for i in buildings}
    k4S, k4I, k4R = deriv(S4, I4, R4)

    # Mise    jour
    for i in buildings:
        buildings[i]["S"] += (dt/6) * (k1S[i] + 2*k2S[i] + 2*k3S[i] + k4S[i])
        buildings[i]["I"] += (dt/6) * (k1I[i] + 2*k2I[i] + 2*k3I[i] + k4I[i])
```

```
        buildings[i]["R"] += (dt/6) * (k1R[i] + 2*k2R[i] +
           2*k3R[i] + k4R[i])

    return buildings
# Calcul sur plusieurs itterations
temps = []
dict_list = []
for t in range(iteration):
    temps.append(temps_o + t*dt)
    dict_list.append(rk4_step(buildings, dt))
    buildings = copy.deepcopy(dict_list[t])

import numpy as np

def construire_matrices(dict_list):
    # Extraire et trier les identifiants des b timents
    cles = sorted(dict_list[0].keys())

    S_matrix = []
    I_matrix = []
    R_matrix = []

    for i in cles:
        ligne_S = [d[i]["S"] for d in dict_list]
        ligne_I = [d[i]["I"] for d in dict_list]
        ligne_R = [d[i]["R"] for d in dict_list]

        S_matrix.append(ligne_S)
        I_matrix.append(ligne_I)
        R_matrix.append(ligne_R)

    return np.array(S_matrix), np.array(I_matrix),
       np.array(R_matrix)

S_matrix, I_matrix, R_matrix = construire_matrices(dict_list)


# Representation de la solution avec ipwidgets
# Th me esth tique
# plt.style.use('seaborn-darkgrid')
matplotlib.rcParams.update({'font.size': 12})


x_values = np.arange(S_matrix.shape[1])

# --- Fonction d'affichage ---
def plot_sir(batiment_index):
    fig, ax = plt.subplots(figsize=(10, 6))

    ax.plot(temps, S_matrix[batiment_index], label="Sains (S)",
       color="#1f77b4", linewidth=2.5)
```

```
    ax.plot(temps, I_matrix[batiment_index], label="Infect s (I)", color="#ff7f0e", linewidth=2.5)
    ax.plot(temps, R_matrix[batiment_index], label="R tablis (R)", color="#2ca02c", linewidth=2.5)

    ax.set_title(f" volution SIR   B timent {batiment_index + 1}", fontsize=15, weight='bold')
    ax.set_xlabel("Temps", fontsize=13)
    ax.set_ylabel("Population", fontsize=13)
    ax.legend(loc="best", fontsize=12)
    ax.grid(True, linestyle='--', alpha=0.6)
    ax.set_ylim(0, max(S_matrix.max(), I_matrix.max(), R_matrix.max()) + 10)

    plt.tight_layout()
    plt.show()

# --- Interface avec slider ---
interact(plot_sir, batiment_index=IntSlider(min=0, max=S_matrix.shape[0]-1, step=1, value=0, description="B timent"))
```

Exemple :

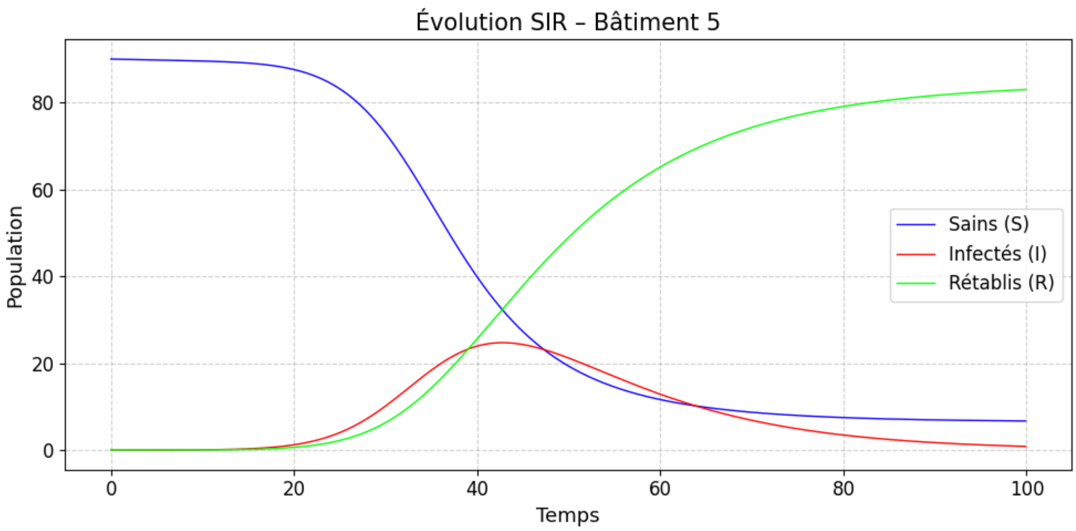

FIGURE 3 – Évolution SIR/ Runge - Kuntta pour le bâtiment 5.

3.4. Conclusion

La simulation montre l’évolution des populations saines, infectées et rétablies dans chaque bâtiment, intégrant diffusion spatiale et dynamique temporelle.

4. Stabilité Numérique et Condition CFL

Nous montrons que la condition CFL pour le couplage EDO-EDP nécessite :

$$\Delta t \leq \min\left(\frac{2}{\beta+\gamma}, \frac{\Delta x^2}{D}\right)$$

4.1. Condition CFL pour l'EDO

Pour la forme EDO du modèle SIR, on a :

$$\frac{dS}{dt} = -\frac{\beta SI}{N}$$
$$\frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

Avec la discrétisation d'Euler explicite, on a :

$$\frac{dX}{dt} \approx \frac{X^{n+1} - X^n}{\Delta t}$$

Calcul pour S

$$S^{n+1} - S^n = -\frac{\beta IS}{N}\Delta t$$
$$S^{n+1} = S^n\left[1 - \frac{\beta I \Delta t}{N}\right]$$

En posant $S^n = G^n$, on a :

$$G = 1 - \frac{\beta \Delta t I}{N}$$

Avec $|G| \leq 1$ et $-1 \leq 1 - \frac{\beta\Delta t I}{N} \leq 1$ et $-2 \leq -\frac{\beta \Delta t I^n}{N} \leq 0$, donc :

$$\Delta t \leq \frac{2}{\beta I/N} \quad (1)$$

Calcul pour I

Par analogie :

$$I^{n+1} = I^n\left[1 + \left(\frac{\beta S}{N} - \gamma\right)\Delta t\right]$$
$$G = 1 + \left(\frac{\beta S}{N} - \gamma\right)\Delta t$$

On a $|G| \leq 1$:

$$-1 \leq 1 + \left(\frac{\beta S}{N} - \gamma\right)\Delta t \leq 1$$

$$-2 \leq \left(\frac{\beta S}{N} - \gamma\right)\Delta t \leq 0$$

Donc :

$$\Delta t \leq \frac{2}{\frac{\beta S}{N} - \gamma} \quad (2)$$

Calcul pour R

De la même manière on a :

$$R^{n+1} = R^n(1 + \gamma\Delta t) \quad \text{et} \quad G = 1 + \gamma\Delta t$$

Avec $|G| \leq 1$ et $-1 \leq 1 + \gamma\Delta t \leq 1$ et $-2 \leq \gamma\Delta t \leq 0$, donc :

$$\Delta t \leq \frac{2}{-\gamma} \quad (3)$$

Pour assurer la stabilité du système, de (2) et (3) on prendra le minimum, soit :

$$\Delta t \leq \min\left(\frac{2}{\beta I/N}, \frac{2}{\frac{\beta S}{N} - \gamma}, \frac{2}{-\gamma}\right)$$

$$\Delta t \leq \frac{2}{\max\left(\frac{\beta I}{N}, \frac{\beta S}{N} - \gamma, -\gamma\right)}$$

Remarquons que :

$$\frac{\beta I}{N} \leq \beta + \gamma, \frac{\beta S}{N} - \gamma \leq \beta + \gamma$$
$$-\gamma < \beta + \gamma \quad \text{car} \quad S \leq N, \quad I \leq N$$

donc :

$$\max\left(\frac{\beta I}{N}, \frac{\beta S}{N} - \gamma, -\gamma\right) \leq \beta + \gamma \quad (4)$$

Ainsi :

$$\Delta t \leq \frac{2}{\beta + \gamma} \quad (4)$$

4.2. Condition CFL pour l'EDP

Pour les EDPs, on ignore les contamination dans le bâtiment, on a donc :

$$\frac{dS}{dt} = D_S \nabla^2 S$$
$$\frac{dI}{dt} = D_I \nabla^2 I$$
$$\frac{dR}{dt} = D_R \nabla^2 R$$

Équations aux Différences

Les trois équations ont aussi la forme :

$$\frac{dX}{dt} = \alpha \nabla^2 X$$

Discrétisation spatiale

Avec la discrétisation par la méthode des volumes finis avec le nombre de voisins égale à 2 en 1D, cela donne même approximation que la différence finie. Par la méthode de la différence finie, on a :

$$\nabla^2 X \approx \frac{X^{n+1} - 2X^n + X^{n-1}}{\Delta x^2}$$

Discrétisation temporelle

Utilisons la méthode d'Euler explicite :

$$\frac{dX}{dt} \approx \frac{X^{n+1} - X^n}{\Delta t}$$

En somme, l'équation discrète est :

$$\frac{X_i^{n+1} - X_i^n}{\Delta t} = \alpha \frac{X_{i+1}^n - 2X_i^n + X_{i-1}^n}{\Delta x^2}$$

Posons

$$X_i^n = G^n e^{ikx_i} = G^n e^{iki\Delta x}$$

On a :

$$\begin{aligned}
Ge^{iki\Delta x} &= \frac{\alpha \Delta t}{\Delta x^2}\left[e^{ik(i+1)\Delta x} + e^{ik(i-1)\Delta x}\right] + \left(1 - 2\alpha\frac{\Delta t}{\Delta x^2}\right) e^{k\Delta x} \\
Ge^{iki\Delta x} &= \frac{\alpha \Delta t}{\Delta x^2}\left(e^{k\Delta x} + e^{-k\Delta x}\right) e^{iki\Delta x} + \left(1 - 2\alpha\frac{\Delta t}{\Delta x^2}\right) e^{iki\Delta x} \\
G &= \frac{\alpha \Delta t}{\Delta x^2}\left[2\cos(k\Delta x)\right] + 1 - 2\alpha\frac{\Delta t}{\Delta x^2} \\
G &= 1 + 2\frac{\alpha \Delta t}{\Delta x^2}\left[-1 + \cos(k\Delta x)\right] \\
G &= 1 - 4\frac{\alpha \Delta t}{\Delta x^2}\left[\sin(\frac{k\Delta x}{2})\right]
\end{aligned}$$

$$\begin{aligned}
\sin\left(\frac{k\Delta x}{2}\right) \leq 1 \quad &\implies 1 - 4\frac{\alpha \Delta t}{\Delta x^2}\left[\sin(\frac{k\Delta x}{2})\right] \geq 1 - 4\frac{\alpha \Delta t}{\Delta x^2} \\
|G| \leq 1 \quad &\implies \quad \left|1 - 4\frac{\alpha \Delta t}{\Delta x^2}\sin\left(\frac{k\Delta x}{2}\right)\right| \leq 1 \\
|G| \leq 1 \quad &\implies \quad \left|1 - 4\frac{\alpha \Delta t}{\Delta x^2}\right| \leq 1 \\
-1 \leq 1 - 4&\frac{\alpha \Delta t}{\Delta x^2} \leq 1 \\
-2 \leq -4&\frac{\alpha \Delta t}{\Delta x^2} \leq 0 \\
\frac{\alpha \Delta t}{\Delta x^2} &\leq \frac{1}{2} \\
\Delta t &\leq \frac{1}{2}\frac{\Delta x^2}{\alpha}
\end{aligned}$$

Pour S on a $\Delta t \leq \frac{1}{2}\frac{\Delta x^2}{D_S}$,
$Pour I$ on a $\Delta t \leq \frac{1}{2}\frac{\Delta x^2}{D_I}$,
$Pour R$ on a $\Delta t \leq \frac{1}{2}\frac{\Delta x^2}{D_R}$

Pour garantir la stabilité du système :

$$\Delta t \leq \frac{1}{2}\frac{\Delta x^2}{\max(D_S, D_I, D_R)}$$

Ainsi :

$$\Delta t \leq \frac{\Delta x^2}{2D} \quad \text{avec} \quad D = \max(D_S, D_I, D_R)$$

Pour vérifier la stabilité dans les deux cas EDO et EDP il faut tel que :

$$\Delta t \leq m$$

Avec m $\leq \frac{2}{\beta+\gamma}, m \leq \frac{\Delta x^2}{2D}$
soit :

$$\Delta t \leq \min\left(\frac{2}{\beta+\gamma}, \frac{\Delta x^2}{2D}\right)$$

Par conséquent, la condition CFL pour le couplage EDO-EDP s'écrit :

$$\Delta t \leq \min\left(\frac{2}{\beta+\gamma}, \frac{\Delta x^2}{2D}\right)$$

5. Analyse de convergence

5.1. Définition de l'Erreur L2

L'erreur L2 mesure la "distance" entre deux solutions sur un ensemble discret de points. Pour deux séquences de valeurs, u_i (par exemple, de la méthode RK4) et v_i (par exemple, de la méthode Euler implicite), à des pas de temps discrets $i = 0, 1, \ldots, N-1$, l'erreur L2 est définie comme :

$$\text{Erreur L2} = \sqrt{\Delta t \sum_{i=0}^{N-1} (u_i - v_i)^2}$$

Algorithme d'implémentation

Algorithm 4 Calcul de l'erreur L2 pour la comparaison des méthodes SIR

Initialiser les matrices S, I, R pour chaque bâtiment i pour les méthodes Euler implicite et RK4
Fixer les paramètres $\beta = 0.3$, $\gamma = 0.1$, $D_S = D_I = D_R = 0.05$, $T = 50$
Définir les listes $\Delta t = [0.1, 0.2, 0.3, 0.4, 0.5]$ et $\Delta x = [0.5, 1.0, 1.5, 2.0, 2.5]$ pour Euler implicite
Fixer $\Delta t_{\text{RK4}} = 0.01$, $\Delta x_{\text{RK4}} = 1.0$ pour RK4
Simuler le modèle SIR avec la méthode Euler implicite
for chaque Δt dans la liste des pas de temps **do**
    Calculer le nombre de pas : steps $= \lfloor T/\Delta t \rfloor$
    for chaque Δx dans la liste des pas spatiaux **do**
        Initialiser les états des bâtiments à partir des conditions initiales
        for chaque pas de temps $t = 0, \Delta t, \ldots, T$ **do**
            Initialiser $S_i^{n+1}, I_i^{n+1}, R_i^{n+1}$ avec les valeurs de n
            repeat
                for chaque bâtiment i **do**
                    Calculer $N_i^{n+1} = S_i^{n+1} + I_i^{n+1} + R_i^{n+1}$
                    Calculer les termes de diffusion : $\sum_{j \in \text{voisins}} (S_j^{n+1} - S_i^{n+1})/\Delta x^2$, et de même pour I et R
                    Mettre à jour S_i^{n+1}, I_i^{n+1}, R_i^{n+1} selon les équations implicites
                end for
            until convergence (variation inférieure à un seuil ϵ)
            Enregistrer l'état dans les matrices S, I, R
        end for
        Stocker les matrices avec la clé $(\Delta t, \Delta x)$
    end for
end for
Simuler le modèle SIR avec la méthode RK4
for chaque pas de temps $t = 0, 0.01, \ldots, T$ **do**
    Calculer les coefficients RK4 $(k1, k2, k3, k4)$ pour S, I, R
    for chaque bâtiment i **do**
        Mettre à jour S_i, I_i, R_i en utilisant la moyenne pondérée des coefficients
    end for
    Enregistrer l'état dans les matrices S, I, R
end for

Calculer les erreurs L2
for chaque type de population (S, I, R) **do**
  for chaque Δx dans la liste des pas spatiaux **do**
    Initialiser une matrice d'erreurs pour les 20 bâtiments et les 5 valeurs de Δt
    for chaque bâtiment $i = 1, 2, \ldots, 20$ **do**
      for chaque Δt dans la liste des pas de temps **do**
        Initialiser la somme des erreurs à 0
        for chaque pas de temps j dans la matrice Euler implicite **do**
          Trouver le pas de temps RK4 correspondant : pos = $\lfloor (j \cdot \Delta t)/0.01 \rfloor$
          Calculer la différence au carré : (valeur Euler[j] − valeur RK4[pos])^2
          Ajouter la différence au carré à la somme des erreurs
        end for
        Calculer l'erreur L2 : $\sqrt{\Delta t \cdot \text{somme des erreurs}}$
        Stocker l'erreur L2 dans la matrice d'erreurs
      end for
    end for
    Stocker la matrice d'erreurs avec la clé Δx
  end for
end for
Visualiser les résultats
Créer des curseurs pour sélectionner Δx et le bâtiment
Afficher un graphique des erreurs L2 pour S, I, R en fonction de Δt

5.1.1. Implémentation en Python

Le code suivant implémente la simulation :

```python
import matplotlib.pyplot as plt
import ipywidgets as widgets
import matplotlib
from ipywidgets import interact, IntSlider, FloatSlider
import numpy as np
import copy

# Initialisation
initial_buildings = {
    1:  {"S": 100, "I": 1,  "R": 0, "neighbors": [2, 3]},
    2:  {"S": 120, "I": 0,  "R": 0, "neighbors": [1, 4]},
    3:  {"S": 80,  "I": 0,  "R": 0, "neighbors": [1, 5]},
    4:  {"S": 150, "I": 0,  "R": 0, "neighbors": [2, 6]},
    5:  {"S": 90,  "I": 0,  "R": 0, "neighbors": [3, 7]},
    6:  {"S": 110, "I": 0,  "R": 0, "neighbors": [4, 8]},
    7:  {"S": 95,  "I": 0,  "R": 0, "neighbors": [5, 9]},
    8:  {"S": 130, "I": 0,  "R": 0, "neighbors": [6, 10]},
    9:  {"S": 85,  "I": 0,  "R": 0, "neighbors": [7, 11]},
    10: {"S": 100, "I": 0,  "R": 0, "neighbors": [8, 12]},
    11: {"S": 75,  "I": 0,  "R": 0, "neighbors": [9, 13]},
    12: {"S": 140, "I": 0,  "R": 0, "neighbors": [10, 14]},
    13: {"S": 105, "I": 0,  "R": 0, "neighbors": [11, 15]},
    14: {"S": 115, "I": 0,  "R": 0, "neighbors": [12, 16]},
```

```
    15: {"S": 125, "I": 0,  "R": 0, "neighbors": [13, 17]},
    16: {"S": 135, "I": 0,  "R": 0, "neighbors": [14, 18]},
    17: {"S": 145, "I": 0,  "R": 0, "neighbors": [15, 19]},
    18: {"S": 155, "I": 0,  "R": 0, "neighbors": [16, 20]},
    19: {"S": 165, "I": 0,  "R": 0, "neighbors": [17]},
    20: {"S": 175, "I": 0,  "R": 0, "neighbors": [18]},
}


        # Debut euler implicite pour plusieur combinason de dt et dx
# Param tres
beta = 0.3
gamma = 0.1
D_S = 0.05
D_I = 0.05
D_R = 0.05
T = 50
To = 0
list_dx = [1/2, 2/2, 3/2, 4/2, 5/2]
list_dt = [0.1*i for i in range(1, 6)]
# D = max(D_I, D_R, D_S)
# list_dx = np.linspace(0.01, 0.1, 101)[:5]
# list_dt = []
# for i in range(5):
#     dx_i = list_dx[i]
#     list_dt.append(round(min(2/(gamma+beta), (dx_i**2)/(2*D)), 5))

for dt in list_dx:
    steps = int(T / dt)
    for dx in list_dx:
        dx2 = dx**2
        def simulate_euler_implicite(buildings_init, steps):
            buildings = copy.deepcopy(buildings_init)
            history = []
            for _ in range(steps):
                S_new = {i: buildings[i]["S"] for i in buildings}
                I_new = {i: buildings[i]["I"] for i in buildings}
                R_new = {i: buildings[i]["R"] for i in buildings}

                for _ in range(20):  # It ration de point fixe
                    S_prev = S_new.copy()
                    I_prev = I_new.copy()
                    R_prev = R_new.copy()

                    for i in buildings:
                        neighbors = buildings[i]["neighbors"]
                        Ni = S_prev[i] + I_prev[i] + R_prev[i]

                        diff_S = sum(S_prev[j] - S_prev[i] for j
```

```
                              in neighbors)
                        diff_I = sum(I_prev[j] - I_prev[i] for j in neighbors)
                        diff_R = sum(R_prev[j] - R_prev[i] for j in neighbors)

                        S_new[i] = buildings[i]["S"] + dt * (-beta * S_prev[i] * I_prev[i] / Ni + D_S * diff_S / dx2)
                        I_new[i] = buildings[i]["I"] + dt * (beta * S_prev[i] * I_prev[i] / Ni - gamma * I_prev[i] + D_I * diff_I / dx2)
                        R_new[i] = buildings[i]["R"] + dt * (gamma * I_prev[i] + D_R * diff_R / dx2)

                for i in buildings:
                    buildings[i]["S"] = S_new[i]
                    buildings[i]["I"] = I_new[i]
                    buildings[i]["R"] = R_new[i]

                history.append(copy.deepcopy(buildings))
            return history

        # Simulation
        history_implicit = simulate_euler_implicite(initial_buildings, steps)

        S_matrix = np.zeros((len(initial_buildings), steps))
        I_matrix = np.zeros((len(initial_buildings), steps))
        R_matrix = np.zeros((len(initial_buildings), steps))
        for i in range(1, 21):
            S_matrix[i-1, :] = [snapshot[i]["S"] for snapshot in history_implicit]
            I_matrix[i-1, :] = [snapshot[i]["I"] for snapshot in history_implicit]
            R_matrix[i-1, :] = [snapshot[i]["R"] for snapshot in history_implicit]

# Global storage for all results
all_S_matrices = {}
all_I_matrices = {}
all_R_matrices = {}

for dt in list_dt:
    steps = int(T / dt)
    for dx in list_dx:
        dx2 = dx**2

        def simulate_euler_implicite(buildings_init, steps):
            buildings = copy.deepcopy(buildings_init)
```

```
    history = []
    for _ in range(steps):
        S_new = {i: buildings[i]["S"] for i in buildings}
        I_new = {i: buildings[i]["I"] for i in buildings}
        R_new = {i: buildings[i]["R"] for i in buildings}

        for _ in range(20):  # Fixed-point iteration
            S_prev = S_new.copy()
            I_prev = I_new.copy()
            R_prev = R_new.copy()

            for i in buildings:
                neighbors = buildings[i]["neighbors"]
                Ni = S_prev[i] + I_prev[i] + R_prev[i]

                diff_S = sum(S_prev[j] - S_prev[i] for j in neighbors)
                diff_I = sum(I_prev[j] - I_prev[i] for j in neighbors)
                diff_R = sum(R_prev[j] - R_prev[i] for j in neighbors)

                S_new[i] = buildings[i]["S"] + dt * (-beta * S_prev[i] * I_prev[i] / Ni + D_S * diff_S / dx2)
                I_new[i] = buildings[i]["I"] + dt * (beta * S_prev[i] * I_prev[i] / Ni - gamma * I_prev[i] + D_I * diff_I / dx2)
                R_new[i] = buildings[i]["R"] + dt * (gamma * I_prev[i] + D_R * diff_R / dx2)

        for i in buildings:
            buildings[i]["S"] = S_new[i]
            buildings[i]["I"] = I_new[i]
            buildings[i]["R"] = R_new[i]

        history.append(copy.deepcopy(buildings))
    return history

# Simulation
history_implicit = simulate_euler_implicite(copy.deepcopy(initial_buildings), steps)

S_matrix = np.zeros((len(initial_buildings), steps))
I_matrix = np.zeros((len(initial_buildings), steps))
R_matrix = np.zeros((len(initial_buildings), steps))

for i in range(1, len(initial_buildings) + 1): # Iterate through building IDs
```

```
            S_matrix[i-1, :] = [snapshot[i]["S"] for snapshot in history_implicit]
            I_matrix[i-1, :] = [snapshot[i]["I"] for snapshot in history_implicit]
            R_matrix[i-1, :] = [snapshot[i]["R"] for snapshot in history_implicit]

        # Store the matrices using a unique key (e.g., a tuple of dt and dx)
        key = (dt, dx)
        all_S_matrices[key] = S_matrix
        all_I_matrices[key] = I_matrix
        all_R_matrices[key] = R_matrix

# Example of how to access a specific matrix:
# If you want the S_matrix for dt=0.1 and dx=1.0:
s_matrix_for_0_1_1_0 = all_S_matrices[(0.1, 1.0)]
print(f"Shape of S_matrix for dt=0.1, dx=1.0: {s_matrix_for_0_1_1_0.shape}")

    # Debut runge kutta
DS = D_S
DI = D_I
DR = D_R

dx = 1.0        # distance spatiale
dt = 0.01        # pas de temps
temps_o = To
temps_total = T  # temps total de simulation
iteration = (temps_total - temps_o)/dt        # pas de temps

def laplacian(var, i, buildings):
    neighbors = buildings[i]["neighbors"]
    return sum((var[j] - var[i]) / dx**2 for j in neighbors)

def rk4_step(buildings, dt):
    def get_states():
        S = {i: buildings[i]["S"] for i in buildings}
        I = {i: buildings[i]["I"] for i in buildings}
        R = {i: buildings[i]["R"] for i in buildings}
        return S, I, R

    S0, I0, R0 = get_states()

    def deriv(S, I, R):
        dS, dI, dR = {}, {}, {}
        for i in buildings:
            N = S[i] + I[i] + R[i]
            dS[i] = -beta * S[i] * I[i] / N + DS * laplacian(S, i, buildings)
            dI[i] = beta * S[i] * I[i] / N - gamma * I[i] + DI *
```

```
                laplacian(I, i, buildings)
            dR[i] = gamma * I[i] + DR * laplacian(R, i, buildings)
        return dS, dI, dR

    # k1
    k1S, k1I, k1R = deriv(S0, I0, R0)

    # k2
    S2 = {i: S0[i] + k1S[i]/2 for i in buildings}
    I2 = {i: I0[i] + k1I[i]/2 for i in buildings}
    R2 = {i: R0[i] + k1R[i]/2 for i in buildings}
    k2S, k2I, k2R = deriv(S2, I2, R2)

    # k3
    S3 = {i: S0[i] + k2S[i]/2 for i in buildings}
    I3 = {i: I0[i] + k2I[i]/2 for i in buildings}
    R3 = {i: R0[i] + k2R[i]/2 for i in buildings}
    k3S, k3I, k3R = deriv(S3, I3, R3)

    # k4
    S4 = {i: S0[i] + k3S[i] for i in buildings}
    I4 = {i: I0[i] + k3I[i] for i in buildings}
    R4 = {i: R0[i] + k3R[i] for i in buildings}
    k4S, k4I, k4R = deriv(S4, I4, R4)

    # Mise   jour
    for i in buildings:
        buildings[i]["S"] += (dt/6) * (k1S[i] + 2*k2S[i] + 2*k3S[i] + k4S[i])
        buildings[i]["I"] += (dt/6) * (k1I[i] + 2*k2I[i] + 2*k3I[i] + k4I[i])
        buildings[i]["R"] += (dt/6) * (k1R[i] + 2*k2R[i] + 2*k3R[i] + k4R[i])

    return buildings

temps = []
dict_list = []
buildings = copy.deepcopy(initial_buildings)
for t in range(int(iteration)):
    temps.append(temps_o + t*dt)
    dict_list.append(rk4_step(buildings, dt))
    buildings = copy.deepcopy(dict_list[t])



def construire_matrices(dict_list):
    # Extraire et trier les identifiants des b timents
    cles = sorted(dict_list[0].keys())

```

```
    S_matrix = []
    I_matrix = []
    R_matrix = []

    for i in cles:
        ligne_S = [d[i]["S"] for d in dict_list]
        ligne_I = [d[i]["I"] for d in dict_list]
        ligne_R = [d[i]["R"] for d in dict_list]

        S_matrix.append(ligne_S)
        I_matrix.append(ligne_I)
        R_matrix.append(ligne_R)

    return np.array(S_matrix), np.array(I_matrix), np.array(R_matrix)

S_matrix, I_matrix, R_matrix = construire_matrices(dict_list)

def return_diff_squared(i, dt, dt_var, dx_var, matrix_rk, big_matrix):
    error_i = 0
    for j in range(big_matrix[(dt_var, dx_var)].shape[1]):
        pos = int((j*dt_var)/dt)
        error_i += (big_matrix[(dt_var, dx_var)][i, j] - matrix_rk[i, pos])**2
    error_i = (dt_var*error_i)**(1/2)
    return error_i

all_error_matrix = np.zeros(len(list_dx))
all_S_error = {}
all_I_error = {}
all_R_error = {}

for matrix_rk, big_matrix, lettre in zip([S_matrix, I_matrix, R_matrix], [all_S_matrices, all_I_matrices, all_R_matrices], ["S", "I", "R"]):
    for j in range(len(list_dx)):
        dx_var = list_dx[j]
        error_matrix = np.zeros((len(initial_buildings), len(list_dt)))
        for i in range(len(initial_buildings)):
            for k in range(len(list_dt)):
                dt_var = list_dt[k]
                error_matrix[i,k] = return_diff_squared(i, dt, dt_var, dx_var, matrix_rk, big_matrix)

        key = (dx_var)
        if lettre == "S":
            all_S_error[key] = error_matrix
        if lettre == "I":
            all_I_error[key] = error_matrix
```

```
        if lettre == "R":
            all_R_error[key] = error_matrix


# X-axis values
x_values = list_dt

# Create sliders
key_slider = widgets.FloatSlider(
    value=0.5,
    min=0.5,
    max=2.5,
    step=0.5,
    description='Delta x:',
    continuous_update=False
)

row_slider = widgets.IntSlider(
    value=0,
    min=0,
    max=19,
    step=1,
    description='Batiment:',
    continuous_update=False
)

# Output widget for the plot
output = widgets.Output()

# Function to update the plot
def update_plot(change):
    with output:
        clear_output(wait=True)
        key = key_slider.value
        row = row_slider.value
        S_values = all_S_error[key][row]
        I_values = all_I_error[key][row]
        R_values = all_R_error[key][row]

        plt.figure(figsize=(15, 5))
        plt.plot(x_values, S_values, label="Sains (S)", color="#0d00ff", linewidth=1)
        plt.plot(x_values, I_values, label="Infect s (I)", color="#ff0000", linewidth=1)
        plt.plot(x_values, R_values, label="R tablis (R)", color="#04ff04", linewidth=1)
        plt.title(f'Batiment {row+1} - Delta {key+1}')
        plt.xlabel('Delta t')
        plt.ylabel('Erreur')
        plt.legend()
        plt.grid(True)
```

```
        plt.show()

# Connect sliders to the update function
key_slider.observe(update_plot, names='value')
row_slider.observe(update_plot, names='value')

# Display sliders and initial plot
display(key_slider, row_slider, output)

# Initial plot
update_plot(None)
```

Exemple :

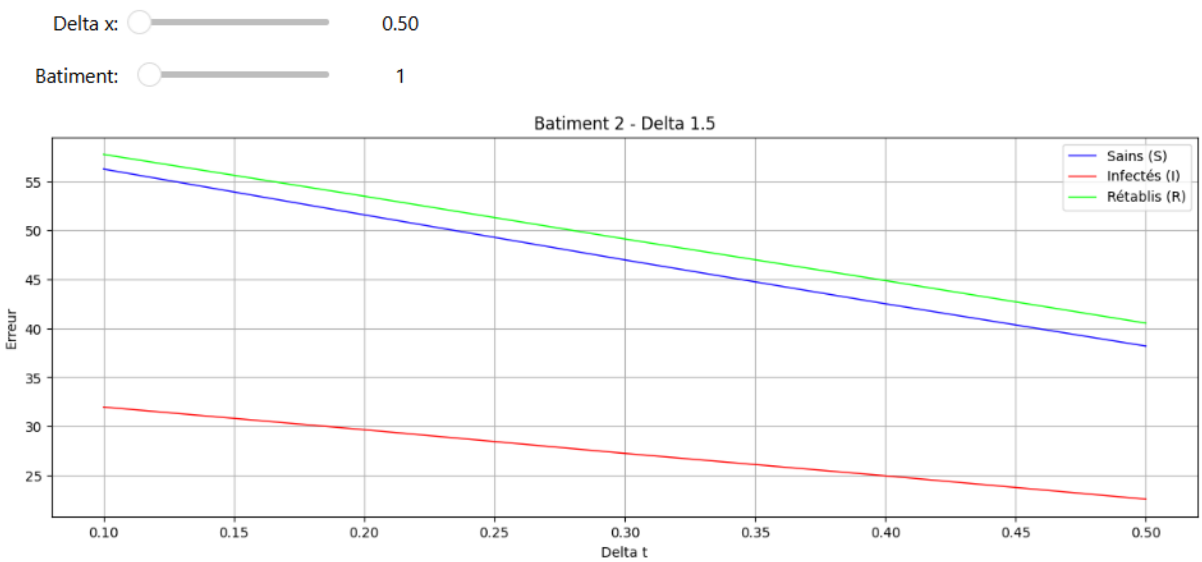

FIGURE 4 – Erreur SIR/ Runge Kuntta - Implicite pour le bâtiment 2 Δt (de 0.10 à 0.50), avec un pas spatial Δx fixé à 0.50.

Les courbes représentent les erreurs pour les populations Sains (bleu), Infectés (rouge) et Rétablis (vert), qui diminuent progressivement à mesure que Δt augmente. Cela indique que l'erreur entre la méthode Euler implicite et la méthode Runge-Kutta 4 devient plus faible avec des pas de temps plus grands, suggérant que les deux méthodes convergent vers des résultats similaires dans cette plage de Δt.

6. Application : adaptation du modèle à des mesures sanitaires localisées

Le modèle SIR spatial présenté précédemment peut être étendu et adapté pour simuler différents scénarios de gestion de l'épidémie sur un campus universitaire. Deux types d'interventions sont particulièrement pertinents dans un tel contexte : l'organisation des cours en présentiel (ou à distance), et l'instauration de confinements localisés dans certains bâtiments. Ces mesures influencent directement les paramètres du modèle, en particulier le taux de transmission β et les coefficients de diffusion D_S, D_I, D_R.

6.1. Cours en présentiel : augmentation locale du taux de transmission

Lorsqu'un bâtiment accueille des cours en présentiel, les interactions entre individus y sont accrues. Cela se traduit, dans le modèle, par une augmentation locale du taux de transmission β. Plus précisément :

- On considère que, pour les bâtiments i où les cours sont maintenus en présentiel, le paramètre β_i est augmenté par rapport à sa valeur de base. Cette augmentation peut être calibrée en fonction du taux d'occupation ou de la densité d'étudiants.
- Mathématiquement, cela revient à introduire un paramètre $\beta_i(t)$ dépendant à la fois du bâtiment i et du temps, permettant de modéliser des changements dans l'organisation pédagogique au fil de l'épidémie.
- Le terme de contamination $\frac{\beta_i(t)S_iI_i}{N_i}$ devient alors spécifique à chaque bâtiment, ce qui introduit une hétérogénéité spatiale dans la propagation.

Cette adaptation permet de simuler des effets localisés tels qu'une flambée épidémique dans un amphithéâtre très fréquenté ou dans une résidence universitaire où les contacts sont nombreux.

6.2. Confinement localisé : suppression de la diffusion inter-bâtiments

Une autre mesure réaliste consiste à restreindre les déplacements entre certains bâtiments (fermeture, isolement, confinement temporaire). Dans le modèle, cela se traduit par une réduction, voire une annulation, des termes de diffusion spatiale.

- Pour un bâtiment confiné i, on impose $D_S^{(i)} = D_I^{(i)} = D_R^{(i)} = 0$, ce qui bloque la diffusion des populations S, I et R vers ou depuis ce bâtiment.
- Le système d'équations devient alors localisé pour ce bâtiment : il évolue indépendamment des autres, en fonction uniquement de sa dynamique interne (contagion et guérison locale).
- Cela permet de simuler des politiques de quarantaine ciblée ou de cloisonnement spatial (zones rouges sur le campus).

Cette modification modélise efficacement l'impact d'une stratégie de containment, et permet d'en analyser l'efficacité : quelle est la propagation de l'épidémie si l'on isole précocement un bâtiment infecté ?

6.3. Vers un modèle réactif et dynamique

Les deux stratégies précédentes peuvent être combinées dans un cadre plus général :

- Chaque paramètre du modèle devient une fonction dépendant du temps et de l'espace : $\beta_i(t)$, $D_S^{(i)}(t)$, etc.

— Des règles de gestion peuvent être implémentées : *si* $I_i(t) > \theta$, *alors* $D^{(i)} \leftarrow 0$ (confinement automatique en cas de dépassement d'un seuil d'alerte).

— Ce cadre permet de simuler des scénarios adaptatifs, proches des réalités de terrain : détection, réaction, adaptation continue.

Résumé

L'adaptabilité du modèle SIR avec diffusion permet de tester numériquement diverses politiques sanitaires ciblées. Grâce à la modulation locale des paramètres β et D, il est possible de simuler l'impact de mesures concrètes telles que les cours en présentiel ou les confinements partiels, d'anticiper les effets systémiques et d'orienter la prise de décision dans un contexte universitaire.

7. Conclusion

En définitive, ce projet nous a permis d'explorer en profondeur la modélisation d'une épidémie sur un campus universitaire à travers un modèle SIR spatial. À partir du modèle mathématique initial, nous avons formalisé un système d'équations différentielles prenant en compte la dynamique des contaminations, des guérisons, mais aussi la mobilité entre les bâtiments.

Pour simuler cette dynamique, nous avons implémenté trois méthodes numériques complémentaires que sont la méthode d'Euler explicite, la méthode d'Euler implicite et la méthode de Runge-Kutta d'ordre 4.

Ces outils nous ont permis de comparer les comportements, les performances et la stabilité de chaque méthode face à un même scénario épidémique.

Ce travail met ainsi en lumière l'intérêt des modèles mathématiques couplés à des méthodes numériques robustes pour comprendre et anticiper l'évolution d'une épidémie dans un environnement structuré. Il ouvre également des perspectives pour intégrer des stratégies sanitaires réalistes, comme des confinements partiels, la fermeture de certains bâtiments ou l'ajustement de la mobilité.

Enfin, cette modélisation constitue une base solide pour des extensions plus complexes, comme l'introduction de paramètres dynamiques, la vaccination ou la prise en compte du comportement humain.

8. Références bibliographiques

1. https://fr.m.wikipedia.org/wiki/Mod
2. https://matplotlib.org/stable/users/index.html
3. https://ipywidgets.readthedocs.io/en/stable/
4. https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.html
5. https://numpy.org/doc/stable/
6. https://docs.python.org/3/library/copy.html
7. Finite Difference Methods for Ordinary and Partial Differential ...