Astruc Lélio
Del Rosso Luca
Edery Nathan

POLYTECH NICE-SOPHIA

Équations de prédation : le modèle de Lotka-Volterra

Présentation du projet

Le sujet aborde la modélisation d'un système proie-prédateur à l'aide du modèle de Lotka-Volterra, ainsi que son implémentation informatique en utilisant des méthodes numériques pour résoudre les équations différentielles. Les équations différentielles représentant l'évolution des populations de proies et de prédateurs sont résolues en utilisant des schémas numériques tels que la méthode d'Euler implicite et la méthode de Runge-Kutta. Les solutions obtenues sont ensuite visualisées graphiquement pour étudier les dynamiques des populations en fonction du temps et des interactions entre elles. Des améliorations potentielles au modèle de Lotka-Volterra sont également proposées et comparées avec les résultats obtenus précédemment. Cette approche permet d'explorer la modélisation mathématique et la simulation informatique d'un système écologique dynamique, et offre des perspectives intéressantes pour comprendre les comportements complexes des populations proie-prédateur dans différents scenarii.

Première partie : modèle à deux équations

Dans cette partie, l'équation notée (1) correspond à l'équation suivante

$$\dot{x} = \alpha x - \beta xy$$
$$\dot{y} = \delta xy - \gamma y$$

où la notation $\dot{x}$ correspond à la dérivée par rapport au temps.

1. Résolution du système avec une méthode d'Euler implicite

La résolution du système en utilisant la méthode d'Euler implicite nous a posé soucis. En effet, nous n'arrivions pas à calculer les termes étant donné que nous avions besoin des termes d'itérations suivants (x_{n+1}) et (y_{n+1}) que nous n'avions pas. Après avoir demandé de l'aide à nos camarades, nous avons compris que nous devions utiliser une méthode de Newton, et ainsi calculer l'inverse de la Jacobienne de la fonction associée au système. Cependant, nous n'avons pas réussi à implémenter cette méthode en `python`. Nous avons donc choisi d'implémenter la méthode d'Euler explicite qui donne des résultats similaires.
Méthode d'Euler explicite :

$$x_{n+1} = x_n + h \times f(x_n, y_n)$$

$$y_{n+1} = y_n + h \times g(x_n, y_n)$$

où $f(x_n, y_n)$ et $g(x_n, y_n)$ sont les expressions discrètes des dérivées de x et y respectivement, évaluées au temps n.
On a ainsi $f(x_n, y_n) = \dot{x}$ et $g(x_n, y_n) = \dot{y}$
Le principe de la résolution numérique avec cette méthode est le suivant :

1. On choisit un vecteur initial (x_0, y_0).
2. On stocke dans les x_n la valeur courante des vecteurs et on itère pour n allant de 1 à notre nombre maximum d'itérations.
3. On incrémente les vecteurs avec la formule énoncée plus tôt ce qui correspond ici à

$$x_n = x_{n-1} + h \times \dot{x}(x_{n-1}, y_{n-1})$$

$$y_n = y_{n-1} + h \times \dot{y}(x_{n-1}, y_{n-1})$$

4. On récupère x_n et y_n

Après avoir fini cette méthode ainsi que la fin du projet ; nous nous sommes repenchés sur la méthode d'Euler implicite avec Newton. Nous avons fini par réussir à l'implémenter en `python` et avons choisi de garder les deux méthodes pour pouvoir les comparer. Vous trouverez en annexes le code de la méthode d'Euler implicite avec Newton.
On définit alors $F : (x, y) \longmapsto \begin{pmatrix} \alpha x - \beta xy \\ \delta xy - \gamma y \end{pmatrix}$
et
$J = \begin{pmatrix} \alpha - \beta y & -\beta x \\ \delta y & \delta x - \gamma \end{pmatrix}$
Ensuite, on peut trouver $x_{n+1}^{k+1} = x_n^k - J^{-1}(x_{n+1}^k)F(x_{n+1}^k)$

2. Résolution du système avec le schéma de Runge-Kutta 4

La méthode de Runge-Kutta d'ordre 4 est une méthode explicite qui utilise les valeurs de la dérivée à différents points à l'intérieur d'un pas de temps pour estimer la nouvelle valeur de la variable à l'instant de temps suivant. Pour appliquer la méthode de Runge-Kutta d'ordre 4, nous avons d'abord reformulé le système d'équations sous la forme d'une fonction vectorielle $F(z)$ avec $z = (x, y)$.
$F : (x, y) \longmapsto \begin{pmatrix} \alpha x - \beta xy \\ \delta xy - \gamma y \end{pmatrix}$
Ensuite, nous avons utilisé la méthode Runge-Kutta pour approximer les solutions x et y du système d'équations différentielles à chaque pas de temps, en suivant l'algorithme suivant

$$z_{n+1} = z_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

avec

$$k_1 = F(t_n, z_n)$$
$$k_2 = F(t_n + \frac{h}{2}, \frac{h}{2}, z_n + \frac{h}{2} * k_1)$$
$$k_3 = F(t_n + \frac{h}{2}, z_n + \frac{h}{2} * k_2)$$
$$k_3 = F(t_n + h, z_n + h * k_3)$$

3. Détermination des points d'équilibre du système

On cherche les points (x^*, y^*) tels que
$\begin{cases} \dot{x}^* = 0 \\ \dot{y}^* = 0 \end{cases}$

Les cas $x = 0$ et $y = 0$ sont des points d'équilibres évidents.
On cherche désormais $\alpha x - \beta xy = 0$

$$\begin{aligned} \alpha x - \beta xy &= 0 \\ x(\alpha - \beta y) &= 0 \\ y &= \frac{\alpha}{\beta} \end{aligned}$$

On cherche maintenant $\delta xy - \gamma y = 0$

$$\begin{aligned} \delta xy - \gamma y &= 0 \\ y(\delta x - \gamma) &= 0 \\ x &= \frac{\gamma}{\delta} \end{aligned}$$

Finalement, on a les points d'équilibres suivants : $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ et $\begin{pmatrix} \frac{\delta}{\gamma} \\ \frac{\alpha}{\gamma} \end{pmatrix}$. Le point $(0, 0)^t$ correspond à l'extinction des deux espèces, tandis que le point $(\frac{\delta}{\gamma}, \frac{\alpha}{\beta})^t$ correspond à la cohabitation entre celles-ci.

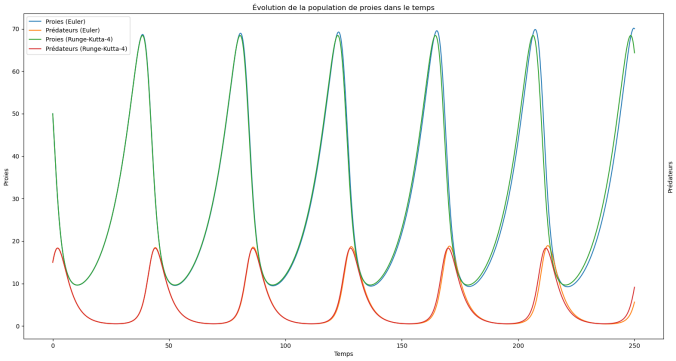
4. Implémentation en Python des schémas numériques

L'implémentation des méthodes numériques en `python` est la partie qui nous a pris le plus de temps. En effet, nous avons rencontré de nombreux problèmes lors de ce travail. Tout d'abord nous n'étions pas sur de comment implémenter les méthodes d'Euler et de Runge-Kutta en dimension 2, puisque nous n'avions jusqu'alors fait que de la dimension 1. Concernant la méthode d'Euler, des camarades nous ont conseillés d'utiliser la méthode de Newton injectée dans la méthode d'Euler, en utilisant l'inverse de la Jacobienne de la fonction f. Nous avons essayé d'implémenter cette méthode mais nous n'avons pas obtenu de résultats satisfaisants et avons préféré passer par la méthode d'Euler explicite.
Lors de l'implémentation de Runge Kutta 4 en `python` nous avons rencontré plusieurs problèmes. Au début nous avions implémenté `RK 4` mais pour chaque équation du système séparemment. Ce qui posait des problèmes lorsque la fonction nous renvoyait des valeurs, car elle ne traitait en entrée que les valeurs d'une seule des équations et pas les deux d'un coup. Ce problème nous empechait d'avoir une bonne simulation de l'évolution du système et c'est le problème qui nous a le plus posé de soucis. Une fois que nous avons corrigé le problème de la fonction `RK4` la simulation fonctionnait correctement. Il a fallu cependant apporter quelques modifications au programme pour avoir des résultats de simulations le plus proche possible des valeurs théoriques.
Le code python est disponible à la fin du rapport, en annexes.

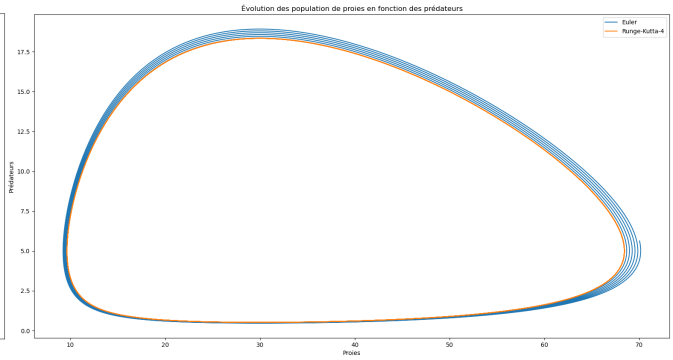
5. Visualisation graphique des méthodes implémentées

Les valeurs des coefficients α, β, γ et δ sont des paramètres à manier avec parcimonie. Une variation, même faible d'un coefficient entraine des résultats très différents. Après avoir essayé

plusieurs valeurs pour chaque coefficient ainsi que des recherches sur internet, nous avons établi des valeurs moyennes pour chaque coefficient. $\alpha \in [0.1, 1]$, $\beta \in [0.01, 0.1]$, $\gamma \in [0.1, 1]$ et $\delta \in [0.01, 0.1]$

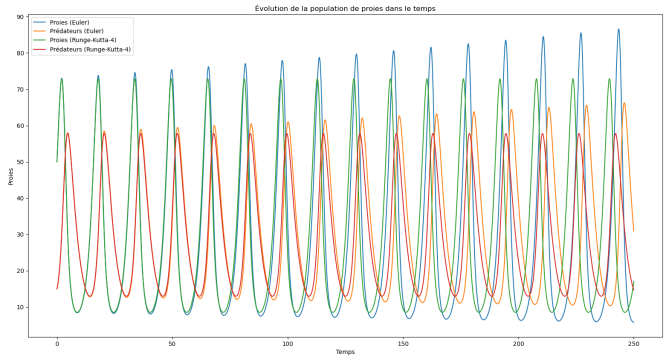


(a) Proies et prédateurs en fonction du temps

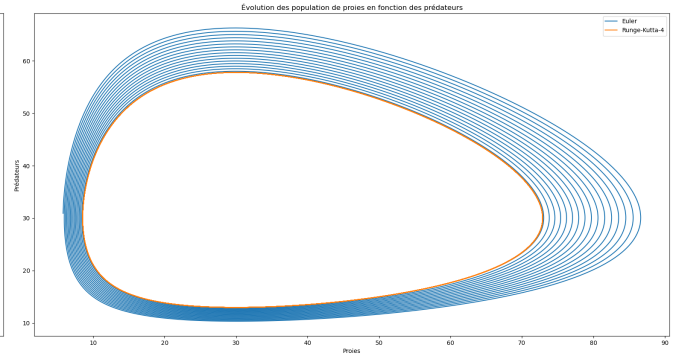


(b) Proies et prédateurs

FIGURE 1 – $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.3$, $\delta = 0.01$, $(x_0, y_0) = (50, 15)$

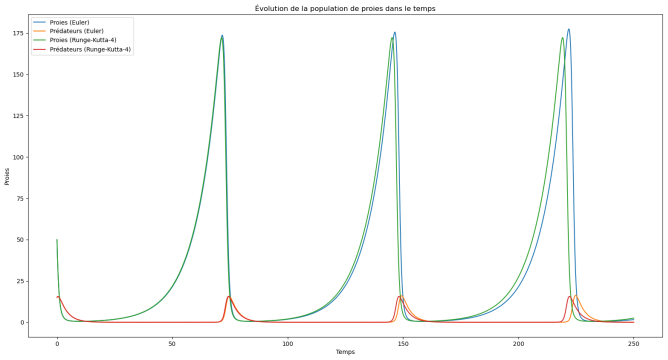


(a) Proies et prédateurs en fonction du temps

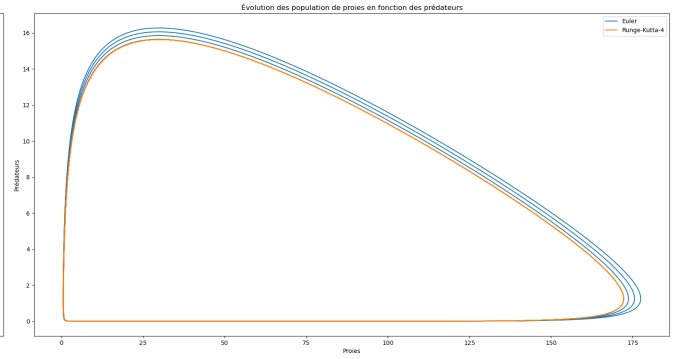


(b) Proies et prédateurs

FIGURE 2 – $\alpha = 0.6$, $\beta = 0.02$, $\gamma = 0.3$, $\delta = 0.01$, $(x_0, y_0) = (50, 15)$

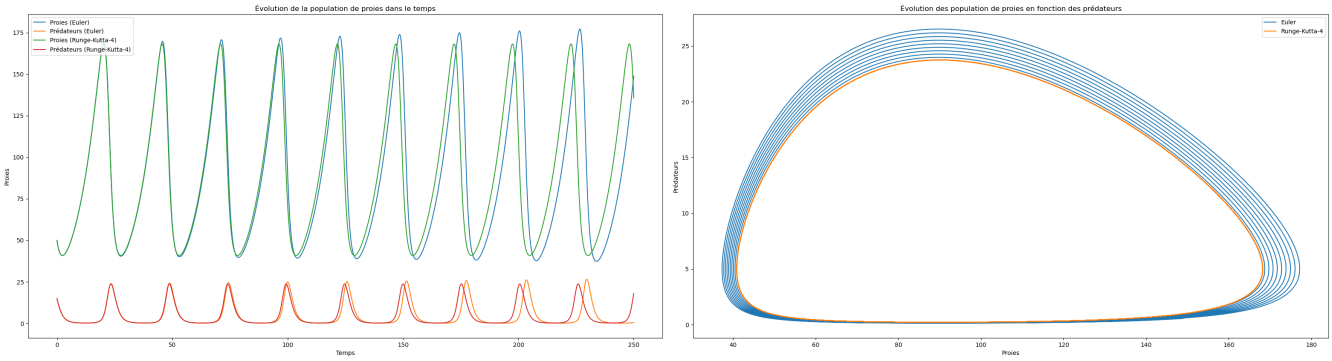


(a) Proies et prédateurs en fonction du temps



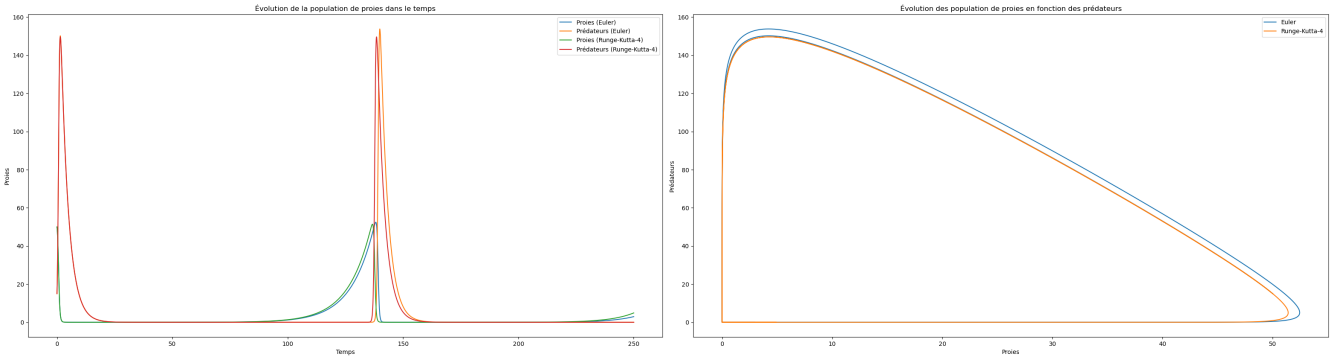
(b) Proies et prédateurs

FIGURE 3 – $\alpha = 0.1$, $\beta = 0.08$, $\gamma = 0.3$, $\delta = 0.01$, $(x_0, y_0) = (50, 15)$



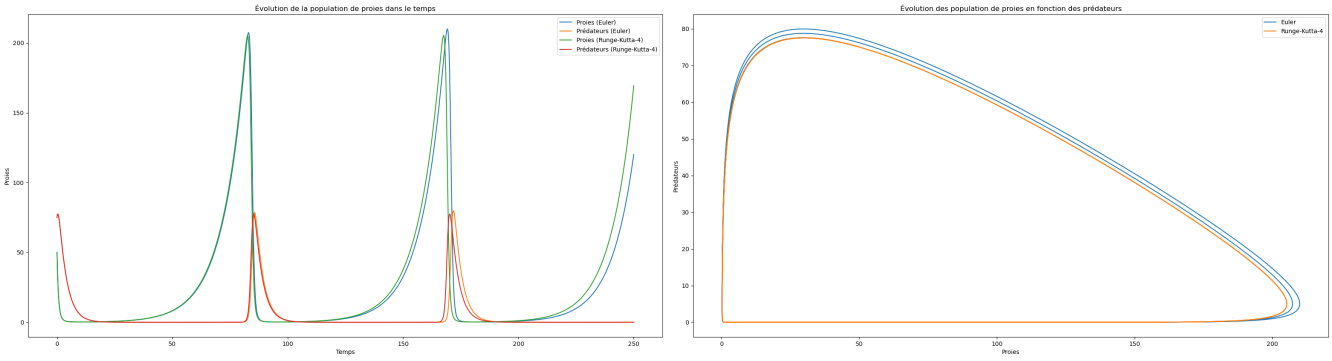
(a) Proies et prédateurs en fonction du temps (b) Proies et prédateurs

FIGURE 4 – $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.9$, $\delta = 0.01$, $(x_0, y_0) = (50, 15)$



(a) Proies et prédateurs en fonction du temps (b) Proies et prédateurs

FIGURE 5 – $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.3$, $\delta = 0.07$, $(x_0, y_0) = (50, 15)$



(a) Proies et prédateurs en fonction du temps (b) Proies et prédateurs

FIGURE 6 – $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.3$, $\delta = 0.01$, $(x_0, y_0) = (50, 75)$

Deuxième partie : modèle à trois équations

6. Modèle à trois population

On considère désormais un modèle à 3 population décrit par le système suivant :

$$\begin{cases} \dot{x} = x(\alpha - \beta x - \gamma y) \\ \dot{y} = y(\delta - \varepsilon y - \zeta x - \eta z) \\ \dot{z} = z(\theta y - \iota z - \kappa) \end{cases}$$

Voici la signification de chaque coefficient :

* α : Taux de croissance des proies en l'absence de prédateurs
* β : Taux de croissances des proies en prenant en compte les ressources
* γ : Interactions entre les proies x et les proies y
* δ : Taux de croissance des proies y en l'absences de prédateurs z
* ε : Taux de croissances des proies y en prenant en compte les ressources
* ζ : Interactions entre les proies y et les proies x
* η : Interactions entre les proies y et les prédateurs z
* θ : Taux de croissance des proies z en l'absence de prédateurs y
* ι : Taux de croissances des prédateurs z en prenant en compte les ressources
* κ : Taux de mortalité naturelle des prédateurs z

1. Méthode de résolution en utilisant une méthode d'Euler explicite

Comme nous l'avons fait dans le modèle à 2 équations, nous pouvons réutiliser la méthode d'Euler explicite avec cette fois ci la formule suivante :

$$x_{n+1} = x_n + h \times f(x_n, y_n, z_n)$$

$$y_{n+1} = y_n + h \times g(x_n, y_n, z_n)$$

$$z_{n+1} = z_n + h \times i(x_n, y_n, z_n)$$

où $f(x_n, y_n, z_n)$, $g(x_n, y_n, z_n)$ et $i(x_n, y_n, z_n)$ sont les expressions discrètes des dérivées de x, y et z respectivement, évaluées au temps n.
On a ainsi $f(x_n, y_n, z_n) = \dot{x}$, $g(x_n, y_n, z_n) = \dot{y}$ et $i(x_n, y_n, z_n) = \dot{z}$
Le principe de la résolution numérique avec cette méthode est le suivant :

1. On choisit un vecteur initial (x_0, y_0, z_0).
2. On stocke dans les x_n la valeur courante des vecteurs et on itère pour n allant de 1 à notre nombre maximum d'itérations.
3. On incrémente les vecteurs avec la formule énoncée plus tôt ce qui correspond ici à

$$x_n = x_{n-1} + h \times \dot{x}(x_{n-1}, y_{n-1}, z_{n-1})$$

$$y_n = y_{n-1} + h \times \dot{y}(x_{n-1}, y_{n-1}, z_{n-1})$$

$$z_n = z_{n-1} + h \times \dot{z}(x_{n-1}, y_{n-1}, z_{n-1})$$

4. On récupère x_n, y_n et z_n

De la même manière que dans le système à deux équations, nous avons réussi juste avant la date butoire à implémenter la méthode d'Euler implicite avec Newton.

2. Méthode de Runge-Kutta

Afin d'appliquer la méthode de Runge-Kutta d'ordre 4 au système d'équations différentielles à trois populations, nous avons d'abord reformulé le système sous la forme d'une fonction vectorielle $G(w)$ avec $w = (x, y, z)$:

$$G : (x, y, z) \longmapsto \begin{pmatrix} x(\alpha - \beta x - \gamma y) \\ y(\delta - \varepsilon y - \zeta x - \eta z) \\ z(\theta y - \iota z - \kappa) \end{pmatrix}$$

Ensuite, nous avons utilisé la méthode Runge-Kutta pour approximer les solutions x, y et z du système d'équations différentielles à chaque pas de temps, en suivant l'algorithme suivant :

$$w_{n+1} = w_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

avec

$$\begin{aligned} k_1 &= G(t_n, w_n) \\ k_2 &= G(t_n + \frac{h}{2}, w_n + \frac{h}{2} \cdot k_1) \\ k_3 &= G(t_n + \frac{h}{2}, w_n + \frac{h}{2} \cdot k_2) \\ k_4 &= G(t_n + h, w_n + h \cdot k_3) \end{aligned}$$

et $w_n = (x_n, y_n, z_n)$.
Cette approche nous a permis d'obtenir des approximations des solutions x, y et z à chaque pas de temps.

3. Calcul des points d'équilibres

On cherche les points du système tels que $\begin{cases} \dot{x} = \alpha x - \beta x^2 - \gamma xy = 0 \\ \dot{y} = \delta y - \varepsilon y^2 - \zeta xy - \eta yz = 0 \\ \dot{z} = \theta zy - \iota z^2 - \kappa z = 0 \end{cases}$

Comme dans le modèle à 2 équations, le point $(0, 0, 0)$ est un point d'équilibre évident. Il faut désormais chercher les points tels que

$$\begin{aligned} \alpha x - \beta x^2 - \gamma xy &= 0 \\ \delta y - \varepsilon y^2 - \zeta xy - \eta yz &= 0 \\ \theta zy - \iota z^2 - \kappa z &= 0 \end{aligned}$$

pour $x, y, z \neq 0$.
Le calcul de ce point d'équilibre est assez conséquent et l'on peut vite se perdre dans la multitude d'inconnues. Aussi, les différents résultats que nous avons chacun trouvés nous ont ralentis dans le calcul de ce point d'équilibre, étant donné que nous ne savions pas qui avait juste et qui avait tort. Voici le point d'équilibre que nous avons trouvé au final.
Soit à calculer :

$$\begin{aligned} \alpha - \beta x - \gamma y &= 0 \\ \delta - \varepsilon y - \zeta x - \eta z &= 0 \\ \theta y - \iota z - \kappa &= 0 \end{aligned}$$

On peut réécrire x et z comme suivent :
$x = \dfrac{\alpha - \gamma y}{\beta}$
$z = \dfrac{\theta y - \kappa}{\iota}$
En remplaçant dans la deuxième équation on trouve

$$\begin{aligned}
&\delta - \varepsilon y - \zeta\left(\frac{\alpha - \gamma y}{\beta}\right) - \eta\left(\frac{\theta y - \kappa}{\iota}\right) = 0 \\
\iff &\delta - \varepsilon y - \frac{\zeta\alpha}{\beta} + \frac{\zeta\gamma y}{\beta} - \frac{\eta\theta y}{\iota} + \frac{\eta\kappa}{\iota} = 0 \\
\iff &-\frac{\varepsilon\beta\iota y}{\beta\iota} + \frac{\zeta\gamma\iota y}{\beta\iota} - \frac{\eta\theta\beta y}{\beta\iota} = -\frac{\delta\beta\iota}{\beta\iota} + \frac{\zeta\alpha\iota}{\beta\iota} - \frac{\eta\kappa\beta}{\beta\iota} \\
\iff &y(\zeta\gamma\iota - \varepsilon\beta\iota - \eta\theta\beta) = -\delta\beta\iota + \zeta\alpha\iota - \eta\kappa\beta \\
\iff &y = \frac{\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa}{\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta}
\end{aligned}$$

Enfin, en remplaçant y dans les expressions de z et de x on trouve

$$x = \frac{\alpha(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta) - \gamma(\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa)}{\beta(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)}$$

$$z = \frac{\theta(\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa) - \kappa(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)}{\iota(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)}$$

Enfin, on trouve comme deuxième point d'équlibre

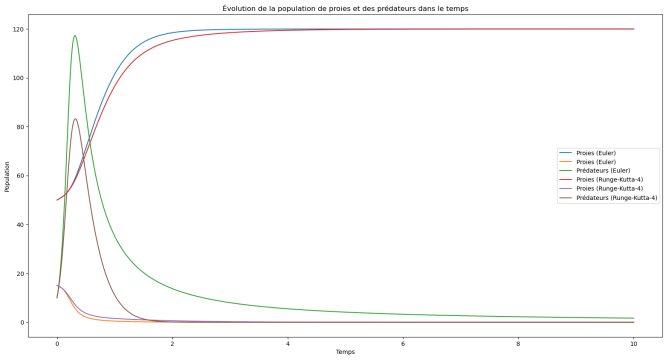
$$\begin{pmatrix} \frac{\alpha(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta) - \gamma(\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa)}{\beta(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)} \\ \frac{\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa}{\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta} \\ \frac{\theta(\alpha\zeta\iota - \beta\delta\iota - \beta\eta\kappa) - \kappa(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)}{\iota(\gamma\zeta\iota - \beta\varepsilon\iota - \beta\eta\theta)} \end{pmatrix}$$

4. Implémentation des méthodes en Python

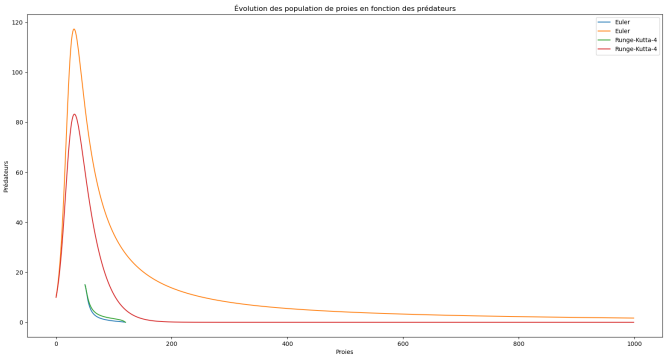
À la différence de l'implémentation sur le modèle à 2 équations, l'implémentation sur ce modèle a été relativement simple, puisque le code est quasiment le même, il suffit seulement de rajouter une ligne de plus et de modifier les dimensions des vecteurs de sortie.

5. Visualisation des résultats

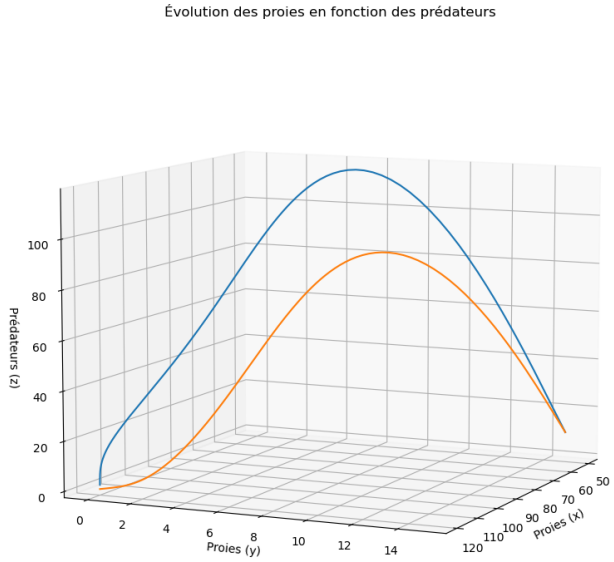
Étant donné le grand nombre de coefficients dans ce système, nous n'allons pas afficher des graphiques pour une modification de chaque coefficient. En effet, il n'est pas très intérressant d'afficher 40 graphes différents, nous serions plus perdus dans les résultats qu'autre chose.



(a) Proies et prédateurs en fonction du temps



(b) Proies et prédateurs



(c) Proies et prédateurs

FIGURE 1 – $\alpha = 3$, $\beta = 0.025$, $\gamma = 0.1$, $\delta = 2$, $\varepsilon = 0.05$, $\zeta = 0.025$, $\eta = 0.05$, $\theta = 1$, $\iota = 0.05$, $\kappa = 0.05$, $(x_0, y_0, z_0) = (50, 15, 10)$

7. Conclusions

Les principales conclusions que nous tiront de ce travail sont les suivantes :

* Les méthodes numériques implémentées sont des méthodes efficaces qui permettent de résoudre le système d'EDO proies-prédateurs ainsi que de visualiser leur évolution dans le temps ou même en fonction d'eux-mêmes.
* Varier les conditions initiales ou les coefficients d'intéractions permettent de prévoir des tendances de populations dans différentes situations.

Nous avons également pu établir quelques unes des limites de ce système :

* Un tel modèle ne prend pas en compte la complexité réelle d'un écosystème tel qu'il en existe sur Terre. En effet, même un système à 10 équations rendrait difficilement compte de la réalité, il faudrait pouvoir inclure des parasytes, des bactéries, des morts naturelles par accidents ou catastrophes naturelles des proies ou des prédateurs.
* Également, ce modèle ne prend pas en compte les conditions géographique. Par exemple, si on modélise le système (1) avec 10 proies et 3 prédateurs ; le temps qu'il faudra pour arriver à un équilibre est différent si les sujets sont placés dans une zone de $10km^2$ ou un continent entier.
* Enfin, même si cela est rare il peut arriver qu'un groupe de proies s'attaque à un prédateur. On pourrait imaginer un groupe d'une dizaine de gazelle contre 1 hyène, si la hyène est blessée, coincée et que les gazelles sont en colère il se pourrait que la hyène soit tuée. Ainsi, si on imagine une modélisation avec un seul prédateur, ce cas ne peux jamais être modélisé dans le modèle de Lotka-Volterra.

8. Perspectives d'améliorations

Forts des constats précédents, voici quelques améliorations auxquelles nous avons pensées :

* Prise en compte des changements climatiques (saisons), ainsi que des potentiels cycle d'hibernation des prédateurs.
* Prise en compte des modifications géographique : par exemple un tremblement de terre, ou, sur une echelle de temps très longue, une séparation des continents.

En conclusion, le modèle de Lotka-Volterra à trois équations, qui prend en compte les interactions entre les populations de proies, de prédateurs et de prédateurs de niveau supérieur, a été étudié et résolu numériquement à l'aide de la méthode d'Euler et de la méthode de Runge-Kutta d'ordre 4. Les solutions obtenues ont été visualisées à l'aide de graphiques, permettant d'observer l'évolution des populations de proies, de prédateurs et de prédateurs de niveau supérieur en fonction du temps. Les résultats obtenus montrent clairement l'impact des interactions complexes entre ces populations, avec des variations périodiques et des comportements dynamiques intéressants. Ce projet a permis de mieux comprendre le comportement du modèle de Lotka-Volterra à trois équations, ainsi que l'importance des méthodes numériques pour résoudre les équations différentielles associées. Il offre également des perspectives pour des recherches futures, telles que l'étude d'autres modèles d'écosystèmes complexes, l'ajout de paramètres pour mieux refléter la réalité biologique, ou encore l'utilisation d'autres méthodes numériques pour améliorer la précision des résultats. En somme, ce projet constitue une approche pratique et enrichissante pour étudier les dynamiques des popula-

tions dans un contexte écologique, et offre des bases solides pour approfondir les recherches dans ce domaine passionnant.

Annexes

Voici en annexes, les codes que nous avons écrits

Modèle à deux équations

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

# Coefficients d interaction entre les populations
alpha = 0.1
beta = 0.02
gamma = 0.3
delta = 0.01

## Conditions initiales
# Population initiale de proies
x0 = 50
# Population initale de predateurs
y0 = 15

#Intervalles de temps, et pas de discretisation h
tempsInitial = 0
tempsFinal = 250
h = 0.01
N = int((tempsFinal - tempsInitial)/h)

def xPoint(x, y):
   return alpha *x - beta *x*y

def yPoint(x, y):
   return delta *x*y - gamma *y

def euler(x0, y0, h, N):
   x = np.zeros(N)
   y = np.zeros(N)
   x[0] = x0
   y[0] = y0
   for n in range(1, N):
       x[n] = x[n-1] + h*xPoint(x[n-1], y[n-1])
       y[n] = y[n-1] + h*yPoint(x[n-1], y[n-1])
   return x, y

x_euler, y_euler = euler(x0, y0, h, N)

nbMaxIterations = 1000
erreur = 1e-6

def F(y, alpha, beta, delta, gamma):
```

```
    x, y = y
    return [alpha * x - beta * x * y, delta * x * y - gamma * y]

def Jacobien(y, alpha, beta, delta, gamma):
    x, y = y
    return [[alpha - beta * y, -beta * x], [delta * y, -gamma + delta * x]]


def eulerNewton(f, j, y0, t0, tf, h, alpha, beta, delta, gamma, nbMaxIterations,
   erreur):
    n = int((tf - t0) / h) + 1
    t = np.linspace(t0, tf, n)
    Y = np.zeros((n, 2))
    Y[0] = y0
    for i in range(1, n):
        yn = Y[i-1].copy()
        for k in range(nbMaxIterations):
            J = np.eye(2) - h * np.array(j(yn, alpha, beta, delta, gamma))
            F = np.array(yn) - np.array(Y[i-1]) - h * np.array(f(yn, alpha, beta,
   delta, gamma))
            dy = np.linalg.solve(J, -F)
            yn += dy
            if np.linalg.norm(dy) < erreur:
                break
        Y[i] = yn
    return t, Y

t, y = eulerNewton(F, Jacobien, y0, tempsInitial, tempsFinal, h, alpha, beta, delta,
   gamma, nbMaxIterations, erreur)

def lotka_volterra(x, y, alpha, beta, gamma, delta):
    dxdt = alpha * x - beta * x * y
    dydt = delta * x * y - gamma * y
    return np.array([dxdt, dydt])

def runge_kutta_4(F, t0, z0, h, N):
    t = np.zeros(N+1)
    z = np.zeros((N+1, len(z0)))
    t[0] = t0
    z[0] = z0

    for n in range(N):
        k1 = F(t[n], z[n])
        k2 = F(t[n] + h/2, z[n] + h/2 * k1)
        k3 = F(t[n] + h/2, z[n] + h/2 * k2)
        k4 = F(t[n] + h, z[n] + h * k3)
        z[n+1] = z[n] + h/6 * (k1 + 2*k2 + 2*k3 + k4)
        t[n+1] = t[n] + h

    return t, z
```

```
F = lambda t, z: lotka_volterra(z[0], z[1], alpha, beta, gamma, delta)

t, z = runge_kutta_4(F, tempsInitial, z0, h, N)

# On extrait les solutions pour x et y
x_rk = z[:, 0]
y_rk = z[:, 1]


min_length = min(len(x_euler), len(x_rk))
x_euler = x_euler[:min_length]
x_rk = x_rk[:min_length]
y_euler = y_euler[:min_length]
y_rk = y_rk[:min_length]


# Evolution de la population de proies et de predateurs dans le temps
plt.figure()
plt.plot(np.linspace(tempsInitial, tempsFinal, N), x_euler, label='Proies (Euler)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), y_euler, label='Predateurs
   (Euler)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), x_rk, label='Proies
   (Runge-Kutta-4)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), y_rk, label='Predateurs
   (Runge-Kutta-4)')
plt.xlabel('Temps')
plt.ylabel('Proies')
plt.legend()
plt.title('Evolution de la population de proies dans le temps')

# Proies en fonction des predateurs
plt.figure()
plt.plot(x_euler, y_euler, label='Euler')
plt.plot(x_rk, y_rk, label='Runge-Kutta-4')
plt.xlabel('Proies')
plt.ylabel('Predateurs')
plt.legend()
plt.title('Evolution des population de proies en fonction des predateurs')
plt.show()
```

Modèle à trois équations

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

# Coefficients d'interaction entre les populations.
```

```
alpha = 3
beta = 0.025
gamma = 0.1
delta = 2
epsilon = 0.05
zeta = 0.025
eta = 0.05
theta = 1
iota = 0.05
kappa = 0.05

# Conditions initiales
# Population initiale de proies
x0 = 50
# Population initale de predateurs
y0 = 15
z0 = 10
w0 = np.array([x0, y0, z0])


#Intervalles de temps, et pas de discretisation h
tempsInitial = 0
tempsFinal = 10
h = 0.01
N = int((tempsFinal - tempsInitial)/h)


def xPoint(x, y):
   return alpha *x - beta *x*x - gamma*x*y

def yPoint(x, y, z):
   return delta*y - epsilon*y*y - zeta*x*y - eta*y*z

def zPoint(y, z):
   return theta*y*z - iota*z*z - kappa*z

def euler(x0, y0, z0, h, N):
   x = np.zeros(N)
   y = np.zeros(N)
   z = np.zeros(N)
   x[0] = x0
   y[0] = y0
   z[0] = z0
   for n in range(1, N):
       x[n] = x[n-1] + h*xPoint(x[n-1], y[n-1])
       y[n] = y[n-1] + h*yPoint(x[n-1], y[n-1], z[n-1])
       z[n] = z[n-1] + h*zPoint(y[n-1], z[n-1])
   return x, y, z

x_euler, y_euler, z_euler = euler(x0, y0, z0, h, N)
```

```
 nbMaxIterations = 1000
erreur = 1e-6

 def F(y, alpha, beta, delta, gamma, epsilon, zeta, eta, theta, iota, kappa):
    x, y, z= y
    return np.array([x * (alpha - beta * x - gamma * y),
           y * (delta - epsilon * y - zeta * x - eta * z),
           z * (theta * y - iota * z - kappa)])

 def Jacobien(y, alpha, beta, delta, gamma, epsilon, zeta, eta, theta, iota, kappa):
    x, y, z = y
    return np.array([[alpha - beta * 2 * x - gamma * y,     -gamma * x,    0],
            [-zeta * y,   delta - epsilon * 2 * y - zeta * x - eta * z,  -eta * y],
            [0,    z * theta,    theta * y - iota * 2 * z - kappa]])


 def eulerNewton(f, j, y0, t0, tf, h, alpha, beta, delta, gamma, epsilon, zeta, eta,
    theta, iota, kappa, nbMaxIterations, erreur):
    n = int((tf - t0) / h) + 1
    t = np.linspace(t0, tf, n)
    Y = np.zeros((n, 3))
    Y[0] = y0
    for i in range(1, n):
        yn = Y[i-1].copy()
        for k in range(nbMaxIterations):
            J = np.eye(3) - h * np.array(j(yn, alpha, beta, delta, gamma, epsilon,
    zeta, eta, theta, iota, kappa))
            F = np.array(yn) - np.array(Y[i-1]) - h * np.array(f(yn, alpha, beta,
    delta, gamma, epsilon, zeta, eta, theta, iota, kappa))
            dy = np.linalg.solve(J, -F)
            yn += dy
            if np.linalg.norm(dy) < erreur:
                break
        Y[i] = yn
    return t, Y

 t, y = eulerNewton(F, Jacobien, y0, tempsInitial, tempsFinal, h, alpha, beta, delta,
    gamma, epsilon, zeta, eta, theta, iota, kappa, nbMaxIterations, erreur)

 def lotka_volterra(x, y, z, alpha, beta, gamma, delta, epsilon, zeta, eta, theta,
    iota, kappa):
    dxdt = x * (alpha - beta * x - gamma * y)
    dydt = y * (delta - epsilon * y - zeta * x - eta * z)

    dzdt = z * (theta * y - iota * z - kappa * x)
    return np.array([dxdt, dydt, dzdt])

 def runge_kutta_4(F, t0, z0, h, N, params):
    t = np.zeros(N+1)
```

```
    z = np.zeros((N+1, len(z0)))
    t[0] = t0
    z[0] = z0

    for n in range(N):
        k1 = F(t[n], z[n], params)
        k2 = F(t[n] + h/2, z[n] + h/2 * k1, params)
        k3 = F(t[n] + h/2, z[n] + h/2 * k2, params)
        k4 = F(t[n] + h, z[n] + h * k3, params)
        z[n+1] = z[n] + h/6 * (k1 + 2*k2 + 2*k3 + k4)
        t[n+1] = t[n] + h

    return t, z

params = (alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota, kappa)
F = lambda t, w, params: lotka_volterra(w[0], w[1], w[2], *params)

# Resoudre le systeme d'equations differentielles
t, w = runge_kutta_4(F, tempsInitial, w0, h, N, params)

# Extraire les solutions pour x, y et z
x_rk = w[:, 0]
y_rk = w[:, 1]
z_rk = w[:, 2]

min_length = min(len(x_euler), len(x_rk))
x_euler = x_euler[:min_length]
x_rk = x_rk[:min_length]
y_euler = y_euler[:min_length]
y_rk = y_rk[:min_length]
z_euler = z_euler[:min_length]
z_rk = z_rk[:min_length]


plt.figure(figsize=(10, 6))
plt.plot(np.linspace(tempsInitial, tempsFinal, N), x_euler, label='Proies (Euler)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), y_euler, label='Proies (Euler)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), z_euler, label='Predateurs
   (Euler)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), x_rk, label='Proies
   (Runge-Kutta-4)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), y_rk, label='Proies
   (Runge-Kutta-4)')
plt.plot(np.linspace(tempsInitial, tempsFinal, N), z_rk, label='Predateurs
   (Runge-Kutta-4)')
plt.xlabel('Temps')
plt.ylabel('Population')
plt.legend()
plt.title('   Evolution de la population de proies dans le temps')
plt.show()
```