



**Master Calcul Haute Performance et Simulation**

**Projet de synthèse**

**Résolution numérique du modèle épidémiologique S.E.I.R.S.  
et modèle multi-agent**

*par Jean-Baptiste Gaillot  
version 1.0*

# Table des matières

<b>1 Résolution numérique du modèle épidémiologique S.E.I.R.S.</b>	<b>1</b>
1.1 Implémentation . . . . .	1
1.2 Analyse des résultats . . . . .	2
1.2.1 Résultats par langage et par méthode . . . . .	2
1.2.2 Résultats par fonction . . . . .	3
1.2.3 Comparaison des langages et des méthodes . . . . .	4
<b>2 Modèle multi-agent</b>	<b>5</b>
2.1 Implémentation . . . . .	5
2.2 Analyse des résultats . . . . .	11
2.2.1 Résultats par langage . . . . .	11
2.2.1.1 Langage Python . . . . .	11
2.2.1.2 Langage C . . . . .	12
2.2.1.3 Langage C++ . . . . .	14
2.2.2 Comparaison des langages et des graines . . . . .	15
2.2.2.1 Comparaison des langages . . . . .	15
2.2.2.2 Comparaison des dates et hauteurs du premier pic en fonction des langages et des graines . . . . .	16
2.2.2.3 Comparaison des temps d'exécution . . . . .	21
2.2.3 Autres résultats à explorer . . . . .	21

## Table des figures

1	Valeur en fonction de la date pour la méthode d'Euler et le langage Python . . . . .	2
2	Valeur en fonction de la date pour la méthode de R.-K. 4 et le langage Python . . . . .	2
3	Valeur en fonction de la date pour la méthode d'Euler et le langage C . . . . .	3
4	Valeur en fonction de la date pour la méthode de R.-K. 4 et le langage C . . . . .	3
5	Valeur en fonction de la date pour la fonction $S$ . . . . .	3
6	Valeur en fonction de la date pour la fonction $E$ . . . . .	3
7	Valeur en fonction de la date pour la fonction $I$ . . . . .	4
8	Valeur en fonction de la date pour la fonction $R$ . . . . .	4
9	Valeur moyenne en fonction de la date . . . . .	11
10	Valeur en fonction de la date pour l'état $S$ . . . . .	12
11	Valeur en fonction de la date pour l'état $E$ . . . . .	12
12	Valeur en fonction de la date pour l'état $I$ . . . . .	12
13	Valeur en fonction de la date pour l'état $R$ . . . . .	12
14	Valeur moyenne en fonction de la date . . . . .	13
15	Valeur en fonction de la date pour l'état $S$ . . . . .	13
16	Valeur en fonction de la date pour l'état $E$ . . . . .	13
17	Valeur en fonction de la date pour l'état $I$ . . . . .	13
18	Valeur en fonction de la date pour l'état $R$ . . . . .	13
19	Valeur moyenne en fonction de la date . . . . .	14
20	Valeur en fonction de la date pour l'état $S$ . . . . .	14
21	Valeur en fonction de la date pour l'état $E$ . . . . .	14
22	Valeur en fonction de la date pour l'état $I$ . . . . .	15
23	Valeur en fonction de la date pour l'état $R$ . . . . .	15
24	Valeur moyenne en fonction de la date pour l'état $S$ . . . . .	15
25	Valeur moyenne en fonction de la date pour l'état $E$ . . . . .	15
26	Valeur moyenne en fonction de la date pour l'état $I$ . . . . .	16
27	Valeur moyenne en fonction de la date pour l'état $R$ . . . . .	16
28	Valeur en fonction de la date pour l'état $I$ et le langage Python . . . . .	17
29	Valeur en fonction de la date pour l'état $I$ et le langage C . . . . .	17
30	Valeur en fonction de la date pour l'état $I$ et le langage C++ . . . . .	17
31	Valeur en fonction de la date pour l'état $I$ . . . . .	18
32	Date du premier pic en fonction du langage . . . . .	18
33	Hauteur du premier pic en fonction du langage . . . . .	18
34	Distribution de la date du premier pic pour le langage Python . . . . .	19
35	Distribution de la date du premier pic pour le langage C . . . . .	19
36	Distribution de la date du premier pic pour le langage C++ . . . . .	19
37	Distribution de la hauteur du premier pic pour le langage Python . . . . .	20
38	Distribution de la hauteur du premier pic pour le langage C . . . . .	20
39	Distribution de la hauteur du premier pic pour le langage C++ . . . . .	20

## Organisation du projet

Lien vers le dépôt GitHub du projet : <https://github.com/gaillot18/M2-CHPS-Projet-synthese.git>

### Informations

- Le but de ce projet est de modéliser une épidémie avec plusieurs méthodes : un modèle épidémiologique et une simulation stochastique.
- On considère chaque méthode comme un problème. Pour chaque problème, on le résout en plusieurs versions, où chaque version représente un langage. La méthode générale de l'implémentation sera la même pour tout les langages, les seules différences sont liées au spécificités du langage (pointeurs, structures, classes, ...). Les noms des fonctions et des variables sont pratiquement les mêmes.
- La machine et l'environnement est le suivant : C.P.U. ARM Apple M3, 16G octets de R.A.M., système conteurisé Linux et système hôte macOS.

### Structure du projet

- Le répertoire **Fonctions-communes** contient des fichiers de fonctions qui sont appelées pour chaque problème (affichages, opérations sur des tableaux, lecture et écritures de tableaux dans un fichier, ...).
- Le répertoire **Problèmes** contient un sous-répertoire par problème. Pour chaque sous-répertoire, il contient les sous-sous-répertoires suivants :
  - Le sous-sous-répertoire **Source** contient les codes sources. Il y a le fichier **resolution.\*** qui contient les fonctions principales de la résolution du problème et le fichier **main.\*** qui est le programme principal qui récupère les arguments de l'exécutable, initialise les variables, appelle la fonction principale de résolution et écrit les résultats dans des fichiers.
  - Le sous-sous-répertoire **Objets** contient les fichiers assembleurs (pour les langages C et C++) .
  - Le sous-sous-répertoire **Librairies** contient les fichiers de définitions des fonctions appelés depuis **main.\*** (pour les langages C et C++) .
  - Le sous-sous-répertoire **Binaires** contient les fichiers exécutables (pour les langages C et C++) .
  - Le sous-sous-répertoire **Resultats-calculation** contient les solutions du problème.
  - Le sous-sous-répertoire **Traitement-resultats** contient ce qui est en rapport avec le traitement des résultats. Il y a le fichier **Notebook.ipynb** qui calcule, affiche et crée des graphiques de résultats. Il stocke les fichiers **csv** et les graphiques.
  - Le fichier **Makefile** compile toutes les versions (pour les langages C et C++) du problème.
  - Le fichier **Script.sh** appelle le Makefile et exécute toutes les versions.
- Le fichier **Script.sh** appelle les scripts de chaque problème.

### Langages de programmation

- Les programmes de résolution sont écrits en Python, C et C++.
- Les programmes principaux sont écrits en Python, C et C++.
- Les programmes contenant les fonctions communes sont écrits en C et C++.
- Les scripts d'exécution sont écrits en Bash (shell).
- Les scripts et règles de compilations sont écrits en make.
- Les programmes de traitements des résultats sont écrits en Python.

# 1 Résolution numérique du modèle épidémiologique S.E.I.R.S.

Soient  $T := ]0, 730]$ ,  $t \in T$  et le système d'équations différentielles ordinaires linéaires suivant : Trouver  $S, E, I, R : T \rightarrow \mathbb{R}$  telles que :

$$\begin{cases} S'(t) = \rho R(t) - \beta I(t) S(t) \\ E'(t) = \beta I(t) S(t) - \sigma E(t) \\ I'(t) = \sigma E(t) - \gamma I(t) \\ R'(t) = \gamma I(t) - \rho R(t) \\ S(0) := 0.999 \\ E(0) := 0 \\ I(0) := 0.001 \\ R(0) := 0 \end{cases}$$

avec

$$\rho := 1/365, \quad \beta := 1/2, \quad \sigma := 1/3 \quad \text{et} \quad \gamma := 1/7.$$

On calculera une solution approchée avec les méthodes (venants de l'analyse numérique) d'Euler et de Runge-Kutta 4.

## 1.1 Implémentation

On crée plusieurs fonctions :

- Fonction pour résoudre les équations pour la méthode d'Euler et le langage Python :

```
def resoudre_equations_euler(T, rho, beta, sigma, gamma, S_0, E_0, I_0, R_0):  
  
    S = np.zeros(T + 1, dtype = np.float64)  
    S[0] = S_0  
    E = np.zeros(T + 1, dtype = np.float64)  
    E[0] = E_0  
    I = np.zeros(T + 1, dtype = np.float64)  
    I[0] = I_0  
    R = np.zeros(T + 1, dtype = np.float64)  
    R[0] = R_0  
    h = 1.0  
  
    for t in range (T):  
        S[t + 1] = S[t] + h * (rho * R[t] - beta * I[t] * S[t])  
        E[t + 1] = E[t] + h * (beta * I[t] * S[t] - sigma * E[t])  
        I[t + 1] = I[t] + h * (sigma * E[t] - gamma * I[t])  
        R[t + 1] = R[t] + h * (gamma * I[t] - rho * R[t])  
  
    return S, E, I, R
```

- Fonction pour résoudre les équations pour la méthode d'Euler et le langage C :

```
void resoudre_equations_euler  
(int T, double rho, double beta, double sigma, double gamma, double S_0,  
double E_0, double I_0, double R_0, double **ptr_S, double **ptr_E,  
double **ptr_I, double **ptr_R){  
  
    double *S;  
    double *E;  
    double *I;  
    double *R;  
    double h;  
  
    S = (double *)calloc(T + 1, sizeof(double));
```

```

S[0] = S_0;
E = (double *)calloc(T + 1, sizeof(double));
E[0] = E_0;
I = (double *)calloc(T + 1, sizeof(double));
I[0] = I_0;
R = (double *)calloc(T + 1, sizeof(double));
R[0] = R_0;
h = 1.0;

for (int t = 0 ; t < T ; t++){
    S[t + 1] = S[t] + h * (rho * R[t] - beta * I[t] * S[t]);
    E[t + 1] = E[t] + h * (beta * I[t] * S[t] - sigma * E[t]);
    I[t + 1] = I[t] + h * (sigma * E[t] - gamma * I[t]);
    R[t + 1] = R[t] + h * (gamma * I[t] - rho * R[t]);
}

*ptr_S = S;
*ptr_E = E;
*ptr_I = I;
*ptr_R = R;

}

```

- Signature de la fonction pour résoudre les équations pour la méthode de Runge-Kutta 4 et le langage Python :

```
resoudre_equations_rungekutta4(T, rho, beta, sigma, gamma, S_0, E_0, I_0, R_0)
```

- Signature de la fonction pour résoudre les équations pour la méthode de Runge-Kutta 4 et le langage C :

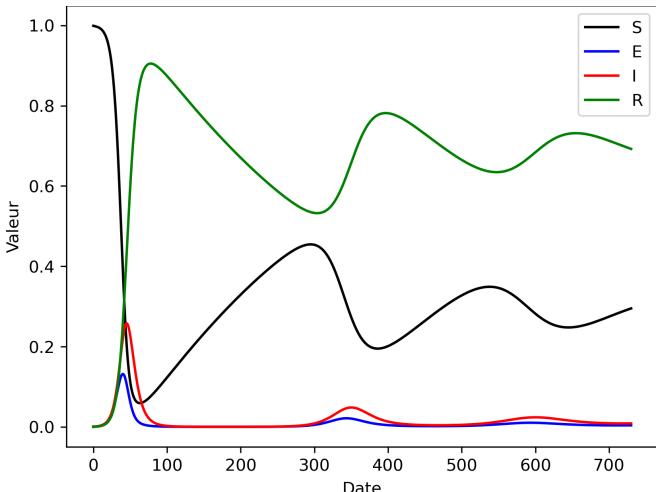
```
void resoudre_equations_rungekutta4
(int T, double rho, double beta, double sigma, double gamma, double S_0,
double E_0, double I_0, double R_0, double **ptr_S, double **ptr_E,
double **ptr_I, double **ptr_R)
```

## 1.2 Analyse des résultats

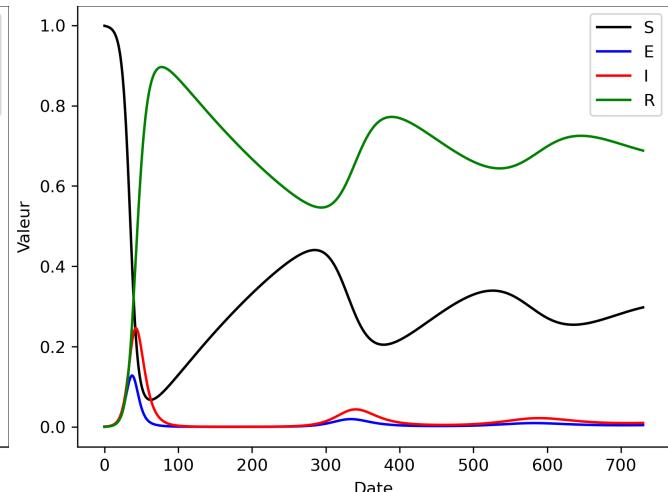
### 1.2.1 Résultats par langage et par méthode

On trace ces graphiques de la valeur en fonction de la date, pour chaque combinaison de langage et méthode :

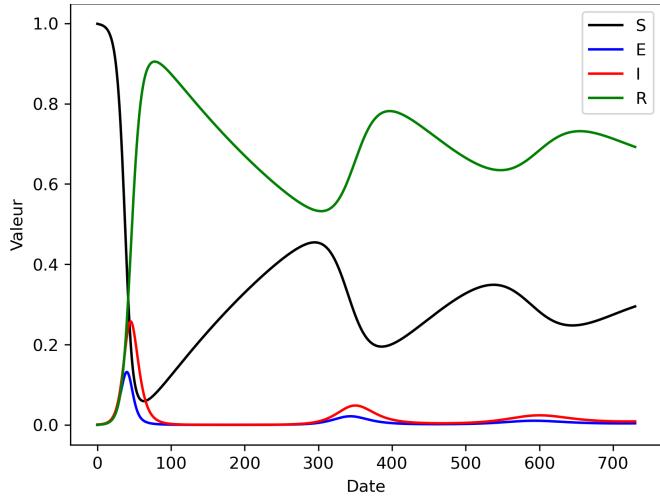
Valeur en fonction de la date  
pour la méthode d'Euler et le langage Python



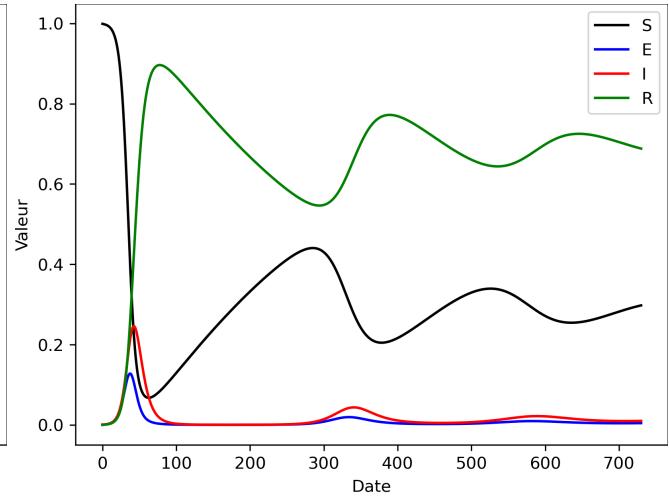
Valeur en fonction de la date  
pour la méthode de R.-K. 4 et le langage Python



Valeur en fonction de la date  
pour la méthode d'Euler et le langage C



Valeur en fonction de la date  
pour la méthode de R.-K. 4 et le langage C

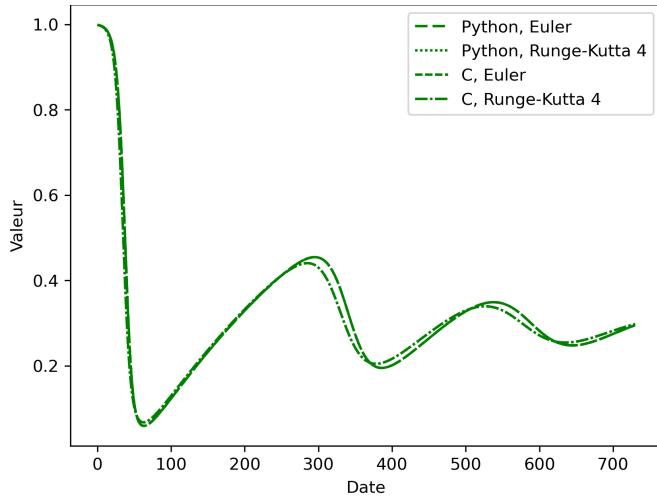


**Commentaire** Pour chaque combinaison de langage et méthode, on a vérifié que la somme (sur les fonctions  $S$ ,  $E$ ,  $I$  et  $R$ ) de la valeur en fonction de la date est égale à 1 pour toute date.

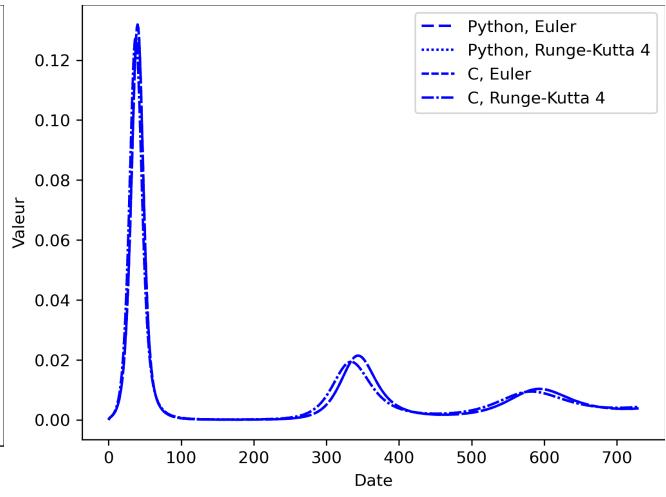
### 1.2.2 Résultats par fonction

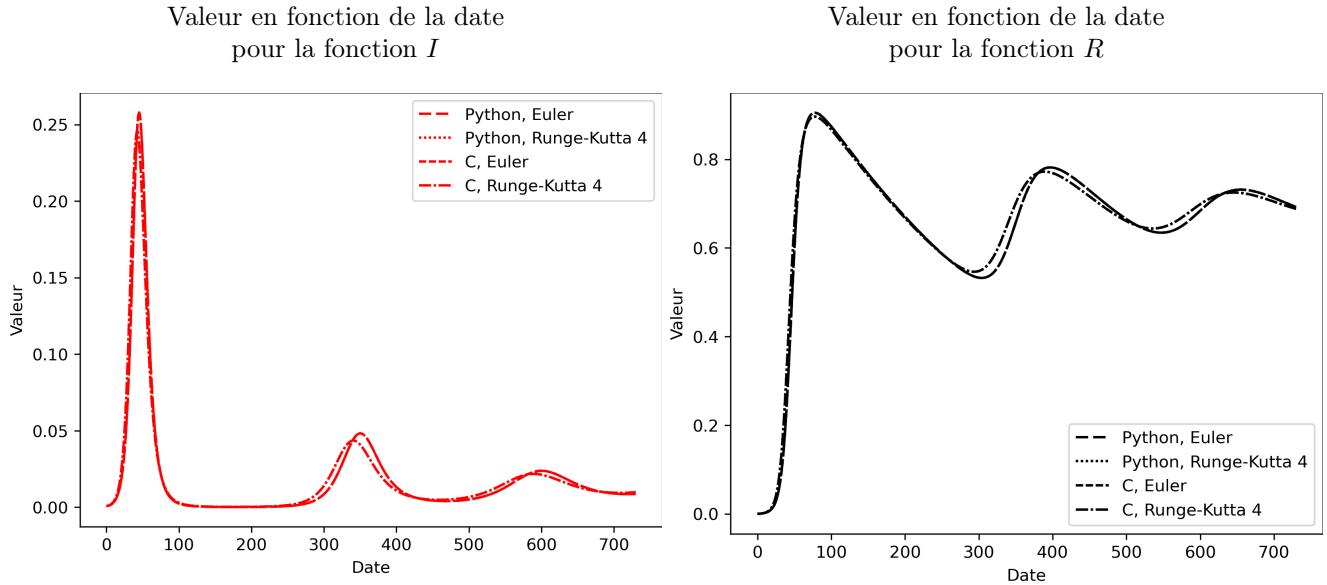
On trace ces graphiques de la valeur en fonction de la date, pour chaque état :

Valeur en fonction de la date  
pour la fonction  $S$



Valeur en fonction de la date  
pour la fonction  $E$





**Commentaire** Graphiquement, pour une méthode identique, les courbes semblent confondues.

### 1.2.3 Comparaison des langages et des méthodes

On note ces résultats de la différence maximale de la valeur en fonction de la date (arrondis à  $10^{-2}$ ) en comparant les langages (en fixant la méthode) et la méthode (en fixant le langage), pour chaque état :

Comparaison	Paramètre fixé	État			
		S	E	I	R
Euler versus Runge-Kutta 4	Langage Python	0.11	0.02	0.04	0.09
Euler versus Runge-Kutta 4	Langage C	0.11	0.02	0.04	0.09
Python versus C	Méthode d'Euler	0.00	0.00	0.00	0.00
Python versus C	Méthode de Runge-Kutta 4	0.00	0.00	0.00	0.00

On note ces résultats de la différence moyenne de la valeur en fonction de la date (arrondis à  $10^{-3}$ ) en comparant les langages (en fixant la méthode) et la méthode (en fixant le langage), pour chaque état :

Comparaison	Paramètre fixé	État			
		S	E	I	R
Euler versus Runge-Kutta 4	Langage Python	0.012	0.001	0.003	0.012
Euler versus Runge-Kutta 4	Langage C	0.012	0.001	0.003	0.012
Python versus C	Méthode d'Euler	0.000	0.000	0.000	0.000
Python versus C	Méthode de Runge-Kutta 4	0.000	0.000	0.000	0.000

### Commentaires

- Pour calculer les différences, on considère tout le vecteur de la valeur en fonction de la date (de taille 731).
- Soient  $u$  et  $v$  des vecteurs de taille  $n$ . Alors, la formule de la différence maximale de  $u$  et  $v$  est la suivante :  $\max_{i \in [1, n]} |u_i - v_i|$  et la formule de la différence moyenne de  $u$  et  $v$  est la suivante :  $\frac{1}{n} \sum_{i=1}^n |u_i - v_i|$ .
- Pour les valeurs par date, il y a des petites différences entre les méthodes (jusqu'à 0.11 pour l'état  $S$ ), mais il n'y a pas de différences entre les langages, ce qui confirme la lecture graphique.
- Le choix du pas de temps est de  $h = 1$  (1 jour) mais on aurait pu utiliser un pas de temps  $h$  plus petit (avec des points intermédiaires entre les jours) pour avoir des résultats un peu plus précis.

## 2 Modèle multi-agent

On résout le même problème que le problème 1 avec une méthode stochastique.

### 2.1 Implémentation

On crée des structures de données (classes pour les langages Python et C++ et structure pour le langage C) pour représenter les propriétés d'un humain :

- Classe `humain` pour le langage Python :

```
class humain:

    def __init__(self):
        self.identifiant = 0
        self.etat = S # S = 0, E = 1, I = 2, R = 3
        self.duree_etat = 0
        self.duree_E = 0.0
        self.duree_I = 0.0
        self.duree_R = 0.0
        self.position_x = 0
        self.position_y = 0
```

- Structure `humain` pour le langage C :

```
struct humain{

    int identifiant;
    int etat; // S = 0, E = 1, I = 2, R = 3
    int duree_etat;
    double duree_E;
    double duree_I;
    double duree_R;
    int position_x;
    int position_y;

};
```

- Classe `humain` pour le langage C++ :

```
class humain{

public:

    int identifiant;
    int etat; // S = 0, E = 1, I = 2, R = 3
    int duree_etat;
    double duree_E;
    double duree_I;
    double duree_R;
    int position_x;
    int position_y;

};
```

#### Commentaires

- Pour les langages C et C++, avec l'alignement mémoire, la classe ou la structure fait 48 octets.
- Des optimisations possibles seraient de stocker l'état avec un type représenté par 2 bits (comme il y a 4 états possibles) et de mieux aligner la mémoire.

On utilise des macros pour les langages C et C++ :

```
# define IDX(i, j, taille_grille) ((i) * (taille_grille) + (j))
# define S 0
# define E 1
# define I 2
# define R 3
```

On crée plusieurs fonctions :

- Fonction pour initialiser les résultats pour le langage C :

```
void initialiser_resultats
(int duree_totale, int **ptr_nb_S, int **ptr_nb_E, int **ptr_nb_I,
int **ptr_nb_R){

    int *nb_S;
    int *nb_E;
    int *nb_I;
    int *nb_R;

    nb_S = (int *)calloc((duree_totale + 1), sizeof(int));
    nb_E = (int *)calloc((duree_totale + 1), sizeof(int));
    nb_I = (int *)calloc((duree_totale + 1), sizeof(int));
    nb_R = (int *)calloc((duree_totale + 1), sizeof(int));

    *ptr_nb_S = nb_S;
    *ptr_nb_E = nb_E;
    *ptr_nb_I = nb_I;
    *ptr_nb_R = nb_R;

}
```

- Fonction pour mettre à jour les tableaux résultats d'une itération pour le langage C :

```
void maj_resultats
(int nb_humains, struct humain *humains, int duree, int *nb_S, int *nb_E,
int *nb_I, int *nb_R){

    for (int identifiant = 0 ; identifiant < nb_humains ; identifiant ++){
        if (humains[identifiant].etat == S){
            nb_S[duree]++;
        }
        if (humains[identifiant].etat == E){
            nb_E[duree]++;
        }
        if (humains[identifiant].etat == I){
            nb_I[duree]++;
        }
        if (humains[identifiant].etat == R){
            nb_R[duree]++;
        }
    }
}
```

- Fonction pour initialiser le tableau d'ordre pour le langage C :

```
void initialiser_ordre(int nb_humains, int **ptr_ordre){

    int *ordre;

    ordre = (int *)malloc(nb_humains * sizeof(int));
```

```

    for (int i = 0 ; i < nb_humains ; i ++){
        ordre[i] = i;
    }

    *ptr_ordre = ordre;

}

```

— Fonction pour mélanger (avec l'algorithme de Fisher-Yates) le tableau d'ordre pour le langage C :

```

void melanger_ordre(int nb_humains, int *ordre){

    for (int i = nb_humains - 1 ; i > 0 ; i --){
        int j;
        int temp;
        j = genrand_int31() % (i + 1);
        temp = ordre[i];
        ordre[i] = ordre[j];
        ordre[j] = temp;
    }

}

```

— Fonction pour initialiser l'ensemble des humains pour le langage C :

```

void initialiser_humains
(int nb_humains, int nb_infectes_initial, int taille_grille,
struct humain **ptr_humains){

    struct humain *humains;

    humains = (struct humain *)malloc(nb_humains * sizeof(struct humain));

    for (int i = 0 ; i < nb_humains ; i ++){
        humains[i].identifiant = i;
        if (i < nb_infectes_initial){
            humains[i].etat = I;
        }
        else{
            humains[i].etat = S;
        }
        humains[i].duree_E = negExp(3.0);
        humains[i].duree_I = negExp(7.0);
        humains[i].duree_R = negExp(365.0);
        humains[i].duree_etat = 0;
        humains[i].position_x = genrand_int31() % taille_grille;
        humains[i].position_y = genrand_int31() % taille_grille;
    }

    *ptr_humains = humains;

}

```

— Fonction pour initialiser la grille d'infectés pour le langage C :

```

void initialiser_grille_infectes
(int nb_humains, struct humain *humains, int taille_grille,
int **ptr_grille_infectes){

    int *grille_infectes;

    grille_infectes = (int *)calloc(taille_grille * taille_grille, sizeof(int));

```

```

    for (int i = 0 ; i < nb_humains ; i ++){
        if (humains[i].etat == I){
            int position_x;
            int position_y;
            position_x = humains[i].position_x;
            position_y = humains[i].position_y;
            grille_infectes[IDX(position_x, position_y, taille_grille)]++;
        }
    }

    *ptr_grille_infectes = grille_infectes;
}

```

- Fonction pour déplacer aléatoirement un humain pour le langage C :

```

void deplacer_humain
(struct humain *humains, int identifiant, int taille_grille,
int *grille_infectes){

    int position_x;
    int position_y;
    int nouvelle_position_x;
    int nouvelle_position_y;

    position_x = humains[identifiant].position_x;
    position_y = humains[identifiant].position_y;
    nouvelle_position_x = genrand_int31() % taille_grille;
    nouvelle_position_y = genrand_int31() % taille_grille;

    humains[identifiant].position_x = nouvelle_position_x;
    humains[identifiant].position_y = nouvelle_position_y;

    if (humains[identifiant].etat == I){
        grille_infectes[IDX(position_x, position_y, taille_grille)]--;
        grille_infectes[IDX(nouvelle_position_x,
                            nouvelle_position_y,
                            taille_grille)]++;
    }
}

```

- Fonction pour vérifier et changer l'état si un humain à l'état S doit passer à l'état E pour le langage C :

```

void exposer
(struct humain *humains, int identifiant, int taille_grille,
int *grille_infectes){

    if (humains[identifiant].etat == S){

        int position_x;
        int position_y;
        int nb_voisins_infectes;
        double probabilite;

        nb_voisins_infectes = 0;
        position_x = humains[identifiant].position_x;
        position_y = humains[identifiant].position_y;

        for (int voisin_x = -1 ; voisin_x <= 1 ; voisin_x++){
            for (int voisin_y = -1 ; voisin_y <= 1 ; voisin_y++){

```

```

        nb_voisins_infectes += grille_infectes[IDX((position_x
            + voisin_x
            + taille_grille)
            % taille_grille,
            (position_y
            + voisin_y
            + taille_grille)
            % taille_grille,
            taille_grille)];
    }
}

probabilite = 1.0 - exp(-0.5 * nb_voisins_infectes);
if (genrand_real2() < probabilite){
    humains[identifiant].etat = E;
    humains[identifiant].duree_etat = 0;
}

}

```

- Fonction pour vérifier et changer l'état si un humain à l'état E doit passer à l'état I pour le langage C :

```

void infecter
(struct humain *humains, int identifiant, int taille_grille,
int *grille_infectes){

    if (humains[identifiant].etat == E
        && (double)humains[identifiant].duree_etat
        > humains[identifiant].duree_E){
        int position_x;
        int position_y;
        position_x = humains[identifiant].position_x;
        position_y = humains[identifiant].position_y;
        humains[identifiant].etat = I;
        grille_infectes[IDX(position_x, position_y, taille_grille)]++;
        humains[identifiant].duree_etat = 0;
    }

}

```

- Fonction pour vérifier et changer l'état si un humain à l'état I doit passer à l'état R pour le langage C :

```

void recuperer
(struct humain *humains, int identifiant, int taille_grille,
int *grille_infectes){

    if (humains[identifiant].etat == I
        && (double)humains[identifiant].duree_etat
        > humains[identifiant].duree_I){
        int position_x;
        int position_y;
        position_x = humains[identifiant].position_x;
        position_y = humains[identifiant].position_y;
        humains[identifiant].etat = R;
        grille_infectes[IDX(position_x, position_y, taille_grille)]--;
        humains[identifiant].duree_etat = 0;
    }

}

```

- Fonction pour vérifier et changer l'état si un humain à l'état R doit passer à l'état S pour le langage C :

```
void sensibiliser(struct humain *humains, int identifiant){

    if (humains[identifiant].etat == R
        && (double)humains[identifiant].duree_etat
        > humains[identifiant].duree_R){
        humains[identifiant].etat = S;
        humains[identifiant].duree_etat = 0;
    }

}
```

- Fonction principale pour le langage C :

```
void simuler
(int nb_humains, int duree_totale, int nb_infectes_initial, int taille_grille,
unsigned long graine, int **ptr_nb_S, int **ptr_nb_E, int **ptr_nb_I,
int **ptr_nb_R){

    struct humain *humains;
    int *grille_infectes;
    int *ordre;
    int *nb_S;
    int *nb_E;
    int *nb_I;
    int *nb_R;

    init_genrand(graine);
    initialiser_resultats(duree_totale, &nb_S, &nb_E, &nb_I, &nb_R);
    initialiser_ordre(nb_humains, &ordre);
    initialiser_humains(nb_humains, nb_infectes_initial, taille_grille, &humains);
    maj_resultats(nb_humains, humains, 0, nb_S, nb_E, nb_I, nb_R);
    initialiser_grille_infectes(nb_humains,
                                humains,
                                taille_grille,
                                grille_infectes);

    for (int duree = 1 ; duree < duree_totale + 1 ; duree ++){
        melanger_ordre(nb_humains, ordre);
        for (int i = 0 ; i < nb_humains ; i ++){
            int identifiant;
            identifiant = ordre[i];
            deplacer_humain(humains, identifiant, taille_grille, grille_infectes);
            exposer(humains, identifiant, taille_grille, grille_infectes);
            infecter(humains, identifiant, taille_grille, grille_infectes);
            recuperer(humains, identifiant, taille_grille, grille_infectes);
            sensibiliser(humains, identifiant);
            humains[identifiant].duree_etat++;
        }
        maj_resultats(nb_humains, humains, duree, nb_S, nb_E, nb_I, nb_R);
    }

    *ptr_nb_S = nb_S;
    *ptr_nb_E = nb_E;
    *ptr_nb_I = nb_I;
    *ptr_nb_R = nb_R;

    free(ordre);
    free(humains);
    free(grille_infectes);
```

## Commentaires

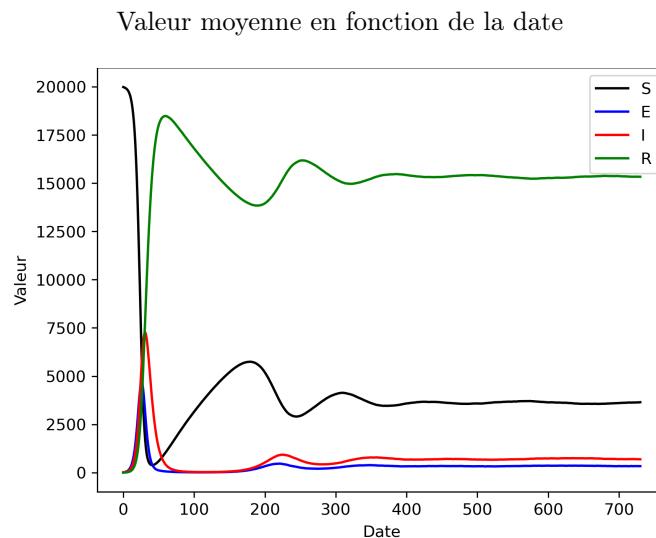
- L'approche est de faire toutes ces fonctions en langage C et ensuite des les adapter en C++ et en Python (en utilisant des classes au lieu des structures, des types `std::Vector` pour le langage C++, ...) avec la même méthode, pour pouvoir avoir la meilleure comparaison statistique possible entre les langages.
- Pour les langages C et C++, les fonctions appelées à chaque itération sont inlinées.
- On utilise un tableau `grille_infectes` (linéarisé en 1D) qui stocke le nombre d'infectés à chaque position. Ce tableau est initialisé au début de la simulation avec les premiers infectés et est mis à jour (par incrémentations ou décrémentations) au cours de la simulation lorsqu'un humain est déplacé, passe d'exposé à infecté ou passe d'infecté à en récupération.
- Pour les langages C et C++, la taille de `*humains` est de  $20000 \cdot 48$  octets = 0.96M octets. Pour le langage Python, la taille de `humains` est difficile à calculer.
- La taille de `grille_infectes` est de  $300 \cdot 300 \cdot 4$  octets = 0.36M octets.
- La taille des `nb_*` où  $*$   $\in \{S, E, I, R\}$  est de  $4 \cdot 731 \cdot 4$  octets = 11.70k octets.
- La taille de `ordre` est de  $713 \cdot 4$  octets = 2.93k octets.
- Dans la fonction `simuler`, après l'initialisation, chaque itération est de la forme suivante : on mélange l'ordre des humains, on les déplace aléatoirement selon cet ordre, on vérifie s'ils doivent passer de sensible à exposé (resp. d'exposer à infecté, d'infecté à en récupération, en récupération à sensible) et on met à jour les résultats.
- Pour générer des nombres pseudo-aléatoires, on utilise le générateur MT.
- Pour le langage C, pour générer des nombres pseudo-aléatoires, on utilise les fonctions reprises sur Internet de MT19937. Pour les langages Python et C++, le générateur MT est directement utilisable.
- Pour chaque langage, on initialise le générateur (avec la graine) une seule fois au début de la fonction `simuler`.
- Pour chaque langage, on fait 30 exécutions avec des graines allant de 0 à 29 par pas de 1.

## 2.2 Analyse des résultats

### 2.2.1 Résultats par langage

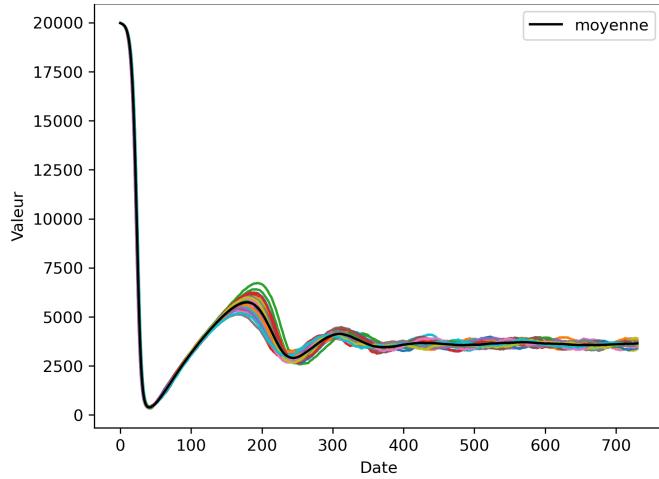
#### 2.2.1.1 Langage Python

On trace ces graphiques de la valeur moyenne (sur les 30 graines) en fonction de la date :

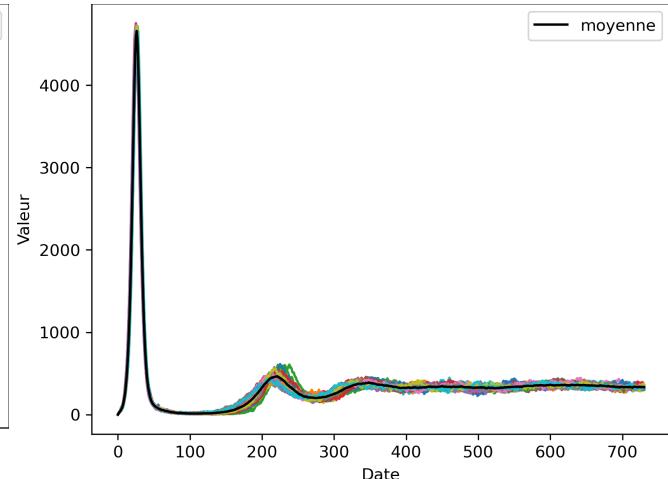


On trace ces graphiques de la valeur (pour les 30 graines) en fonction de la date, pour chaque état :

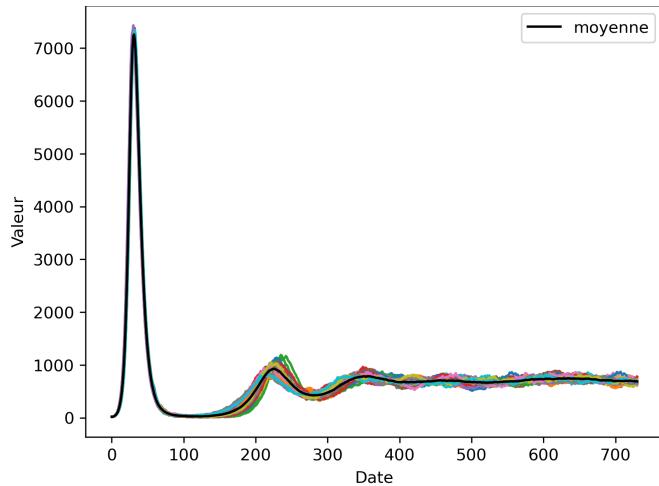
Valeur en fonction de la date  
pour l'état  $S$



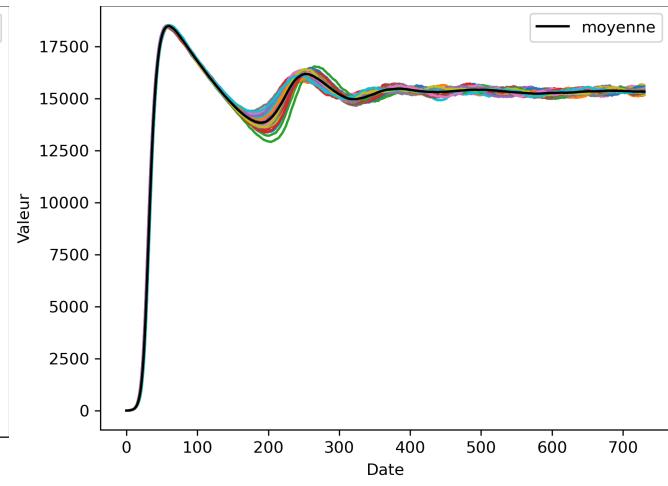
Valeur en fonction de la date  
pour l'état  $E$



Valeur en fonction de la date  
pour l'état  $I$



Valeur en fonction de la date  
pour l'état  $R$

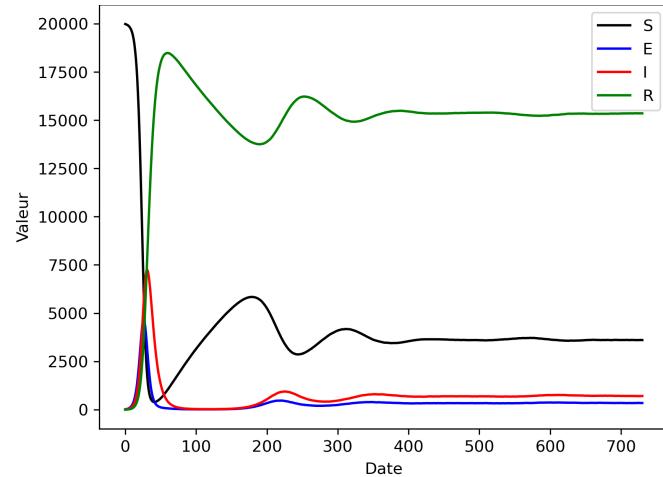


**Commentaire** On a vérifié que la somme (sur les états  $S$ ,  $E$ ,  $I$  et  $R$ ) de la valeur moyenne (sur les 30 graines) en fonction de la date est égale à 20000 pour toute date.

### 2.2.1.2 Langage C

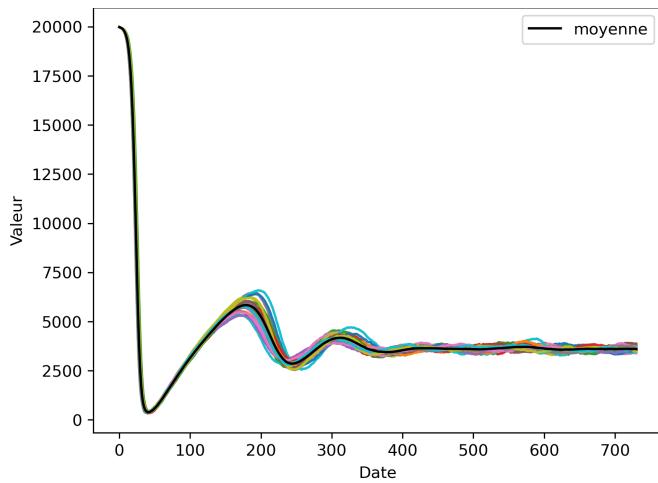
On trace ces graphiques de la valeur moyenne (sur les 30 graines) en fonction de la date :

Valeur moyenne en fonction de la date

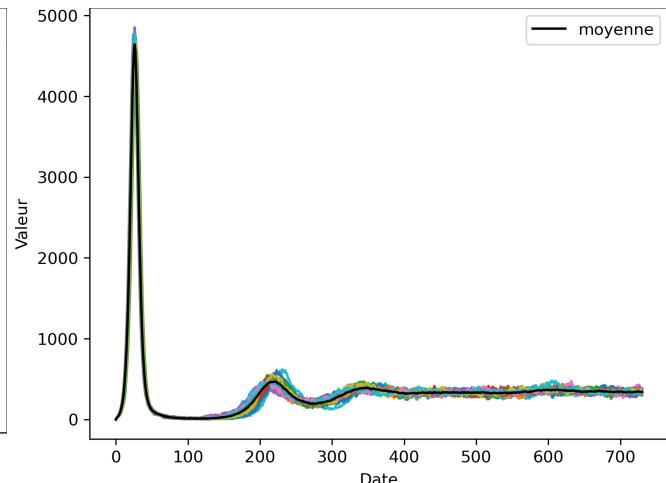


On trace ces graphiques de la valeur (pour les 30 graines) en fonction de la date, pour chaque état :

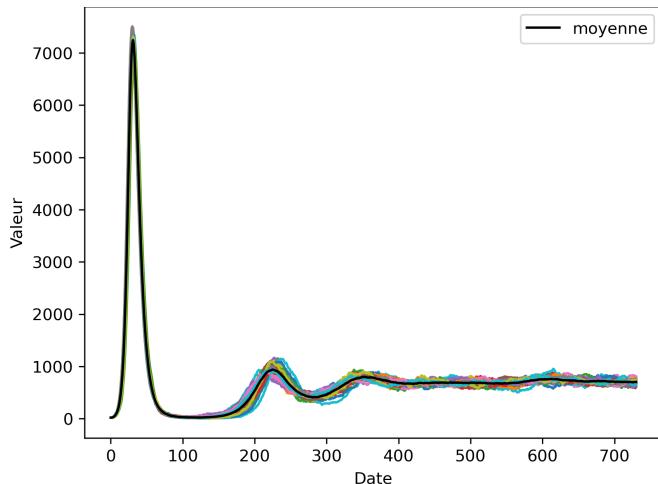
Valeur en fonction de la date  
pour l'état  $S$



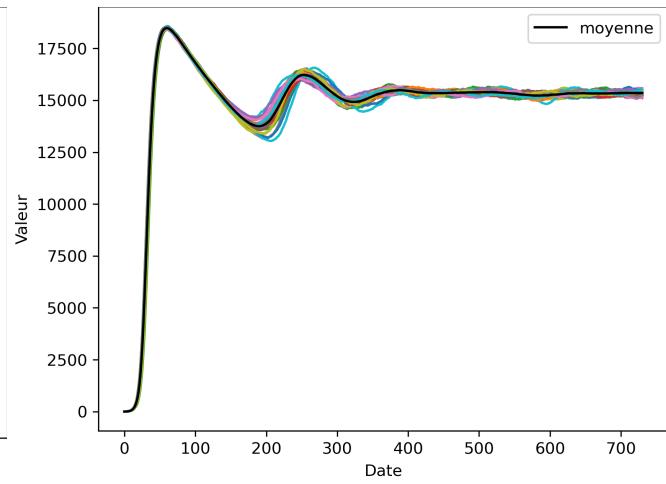
Valeur en fonction de la date  
pour l'état  $E$



Valeur en fonction de la date  
pour l'état  $I$



Valeur en fonction de la date  
pour l'état  $R$

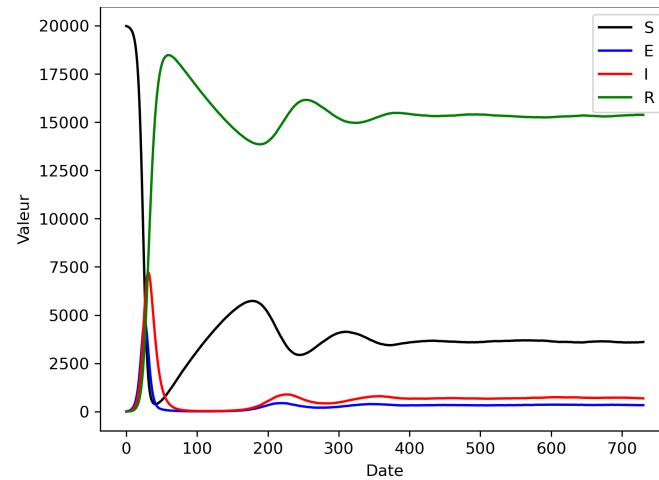


**Commentaire** On a vérifié que la somme (sur les états  $S$ ,  $E$ ,  $I$  et  $R$ ) de la valeur moyenne (sur les 30 graines) en fonction de la date est égale à 20000 pour toute date.

### 2.2.1.3 Langage C++

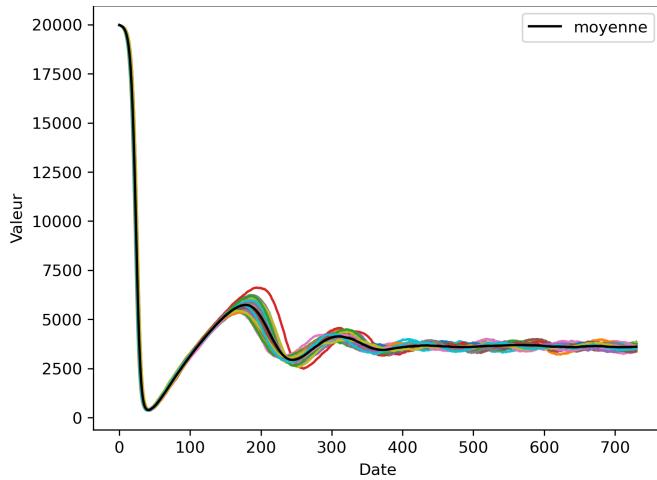
On trace ces graphiques de la valeur moyenne (sur les 30 graines) en fonction de la date :

Valeur moyenne en fonction de la date

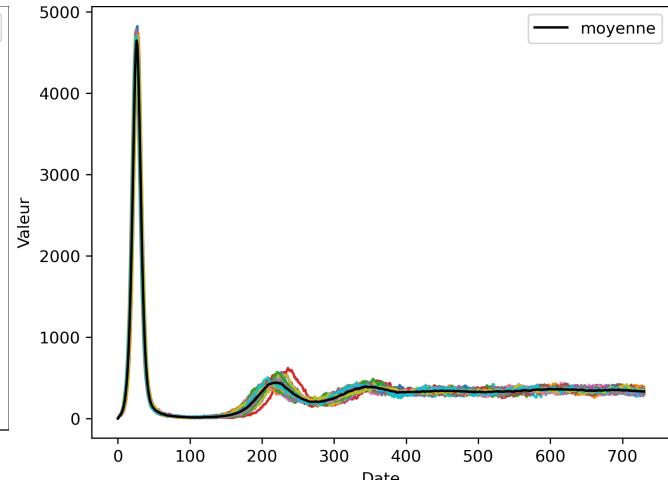


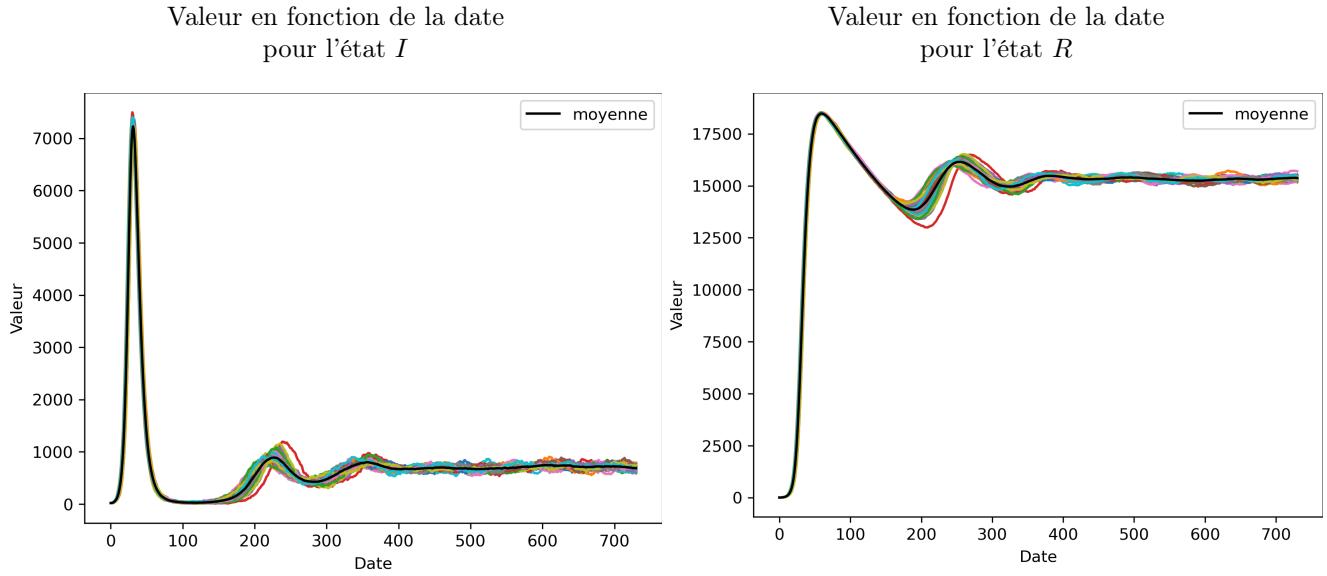
On trace ces graphiques de la valeur (pour les 30 graines) en fonction de la date, pour chaque état :

Valeur en fonction de la date  
pour l'état  $S$



Valeur en fonction de la date  
pour l'état  $E$



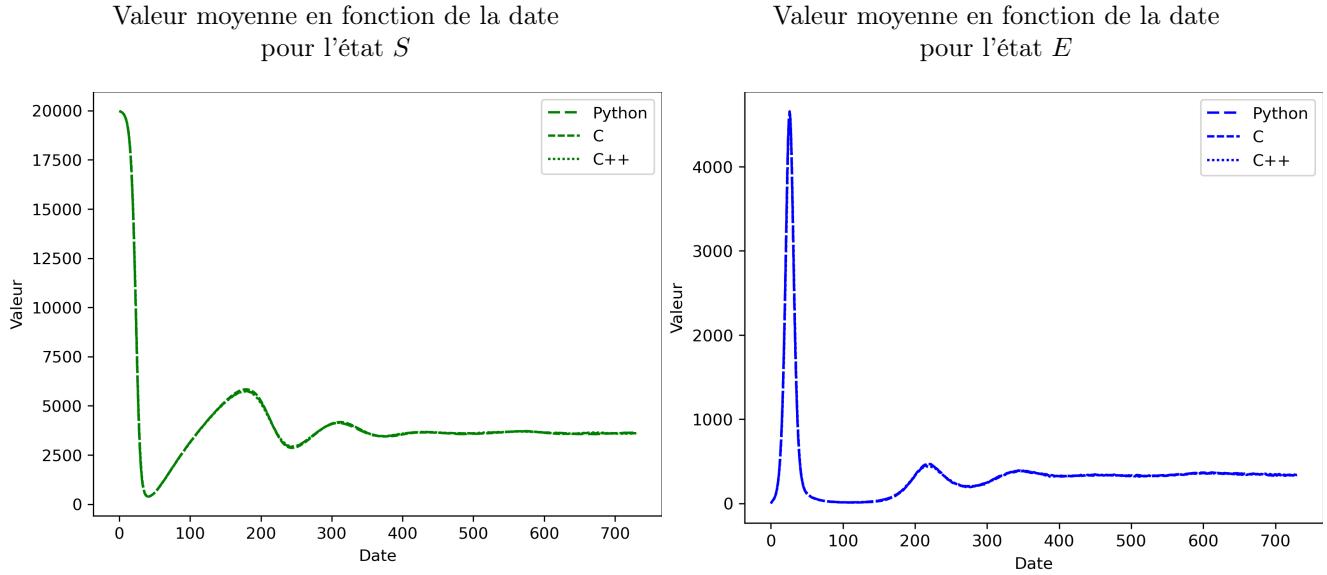


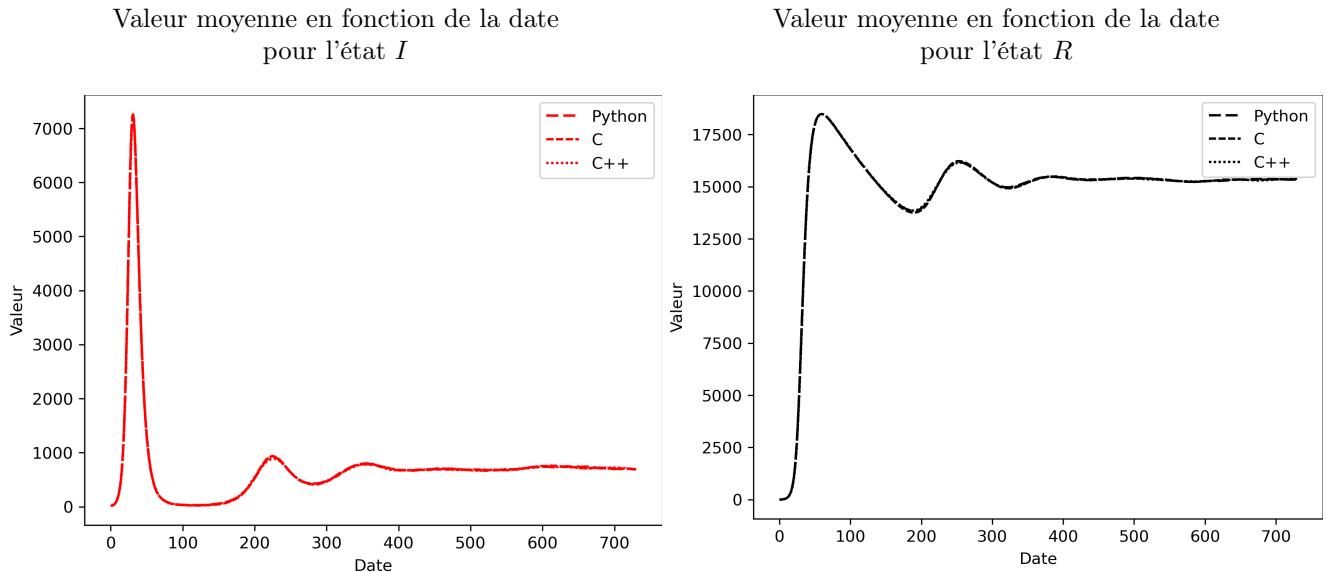
**Commentaire** On a vérifié que la somme (sur les états  $S$ ,  $E$ ,  $I$  et  $R$ ) de la valeur moyenne (sur les 30 graines) en fonction de la date est égale à 20000 pour toute date.

## 2.2.2 Comparaison des langages et des graines

### 2.2.2.1 Comparaison des langages

On trace ces graphiques de la valeur moyenne (pour les 30 graines) en fonction de la date, pour chaque état :





On note ces résultats de la différence maximale de la valeur en fonction de la date (arrondis à  $10^0$ ) en comparant les langages, pour chaque état :

Comparaison	État			
	$S$	$E$	$I$	$R$
Python versus C	159	63	94	115
C versus C++	256	96	106	152
Python versus C++	391	150	200	256
Moyenne	269	103	133	175

On note ces résultats de la différence moyenne de la valeur en fonction de la date (arrondis à  $10^0$ ) en comparant les langages, pour chaque état :

Comparaison	État			
	$S$	$E$	$I$	$R$
Python versus C	29	6	11	27
C versus C++	32	7	12	29
Python versus C++	27	8	12	25
Moyenne	29	7	12	27

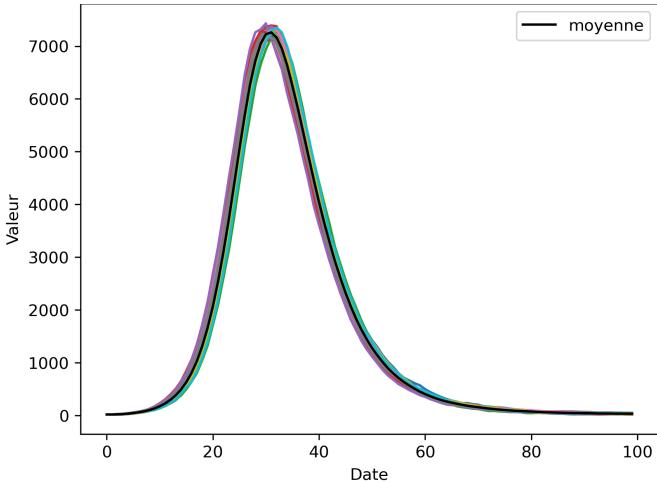
### Commentaires

- Les variations de la valeur en fonction de la date entre les langages sont faibles.
- L'état qui présente la plus grande différence de la valeur en fonction de la date entre les langages est  $S$  (avec 269 de différence maximale moyenne et 29 de différence moyenne moyenne).
- L'état qui présente la plus petite différence de la valeur en fonction de la date entre les langages est  $E$  (avec 103 de différence maximale moyenne et 7 de différence moyenne moyenne).

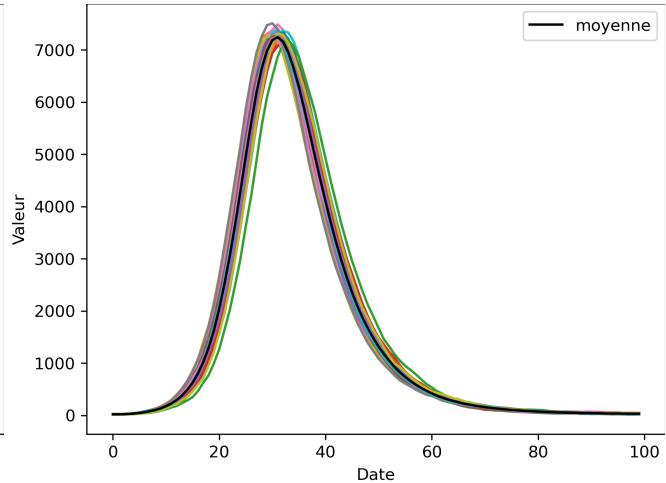
#### 2.2.2.2 Comparaison des dates et hauteurs du premier pic en fonction des langages et des graines

On trace ces graphiques (zoomés entre les jours 0 et 100, pour voir le premier pic) de la valeur en fonction de la date pour l'état  $I$ , pour chaque langage :

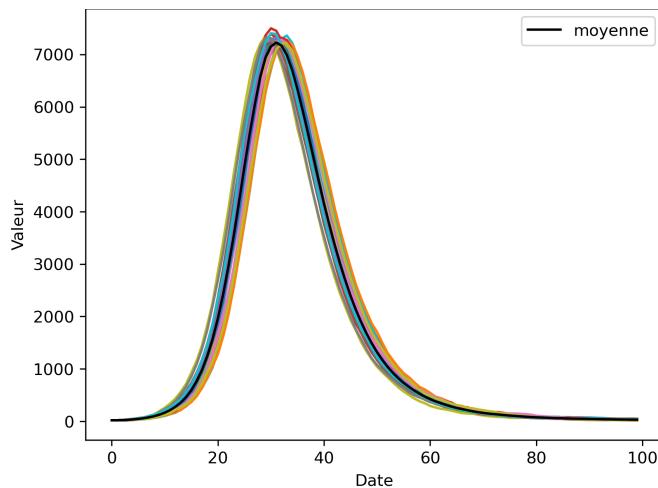
Valeur en fonction de la date  
pour l'état  $I$  et le langage Python



Valeur en fonction de la date  
pour l'état  $I$  et le langage C

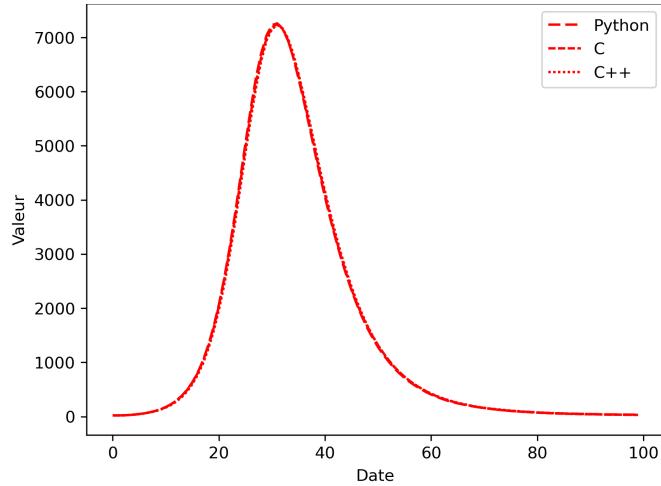


Valeur en fonction de la date  
pour l'état  $I$  et le langage C++



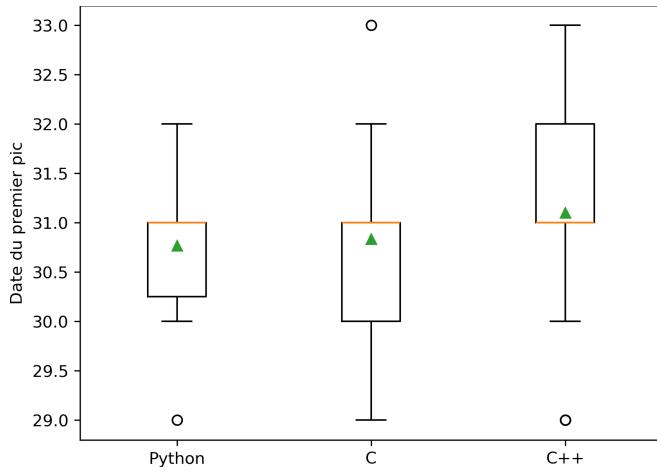
On trace ce graphique (zoomé entre les jours 0 et 100, pour voir le premier pic) de la valeur moyenne (pour les 30 graines) en fonction de la date pour l'état  $I$  :

Valeur moyenne en fonction de la date  
pour l'état  $I$

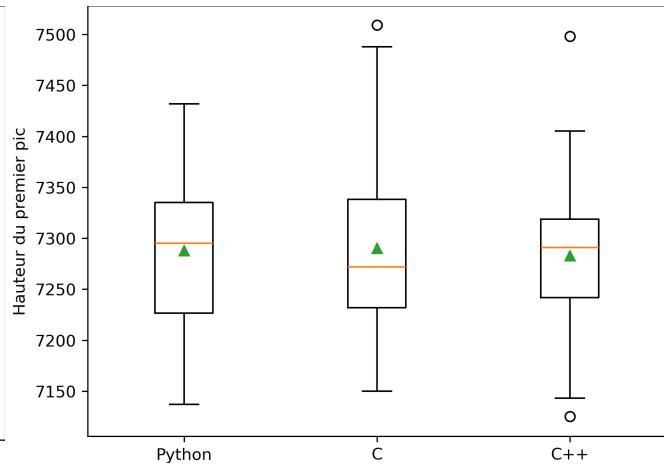


On trace ces graphiques (en boîte) de statistiques de la date et hauteur du premier pic :

Date du premier pic  
en fonction du langage



Hauteur du premier pic  
en fonction du langage



On note ces statistiques de la date du premier pic (arrondies à  $10^{-1}$ ) en fonction du langage :

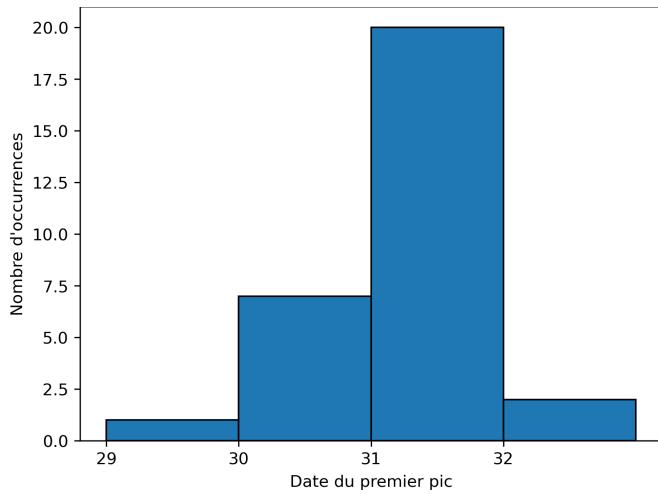
Type de statistique	Langage		
	Python	C	C++
Moyenne	30.8	30.8	31.1
Minimum	29.0	29.0	29.0
Quartile 0.25	30.3	30.0	31.0
Quartile 0.50	31.0	31.0	31.0
Quartile 0.75	31.0	31.0	32.0
Maximum	32.0	33.0	33.0
Écart-type	0.6	0.9	1.0

On note ces statistiques de la hauteur du premier pic (arrondies à  $10^0$ ) en fonction du langage :

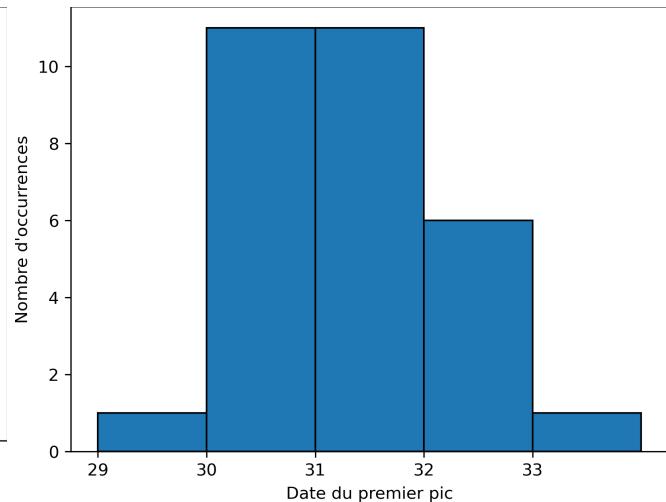
Type de statistique	Langage		
	Python	C	C++
Moyenne	7288	7290	7283
Minimum	7137	7150	7125
Quartile 0.25	7227	7232	7242
Quartile 0.50	7295	7272	7291
Quartile 0.75	7335	7338	7319
Maximum	7432	7509	7498
Écart-type	72	85	79

On trace ces graphiques (en histogramme) de la distribution de la date du premier pic :

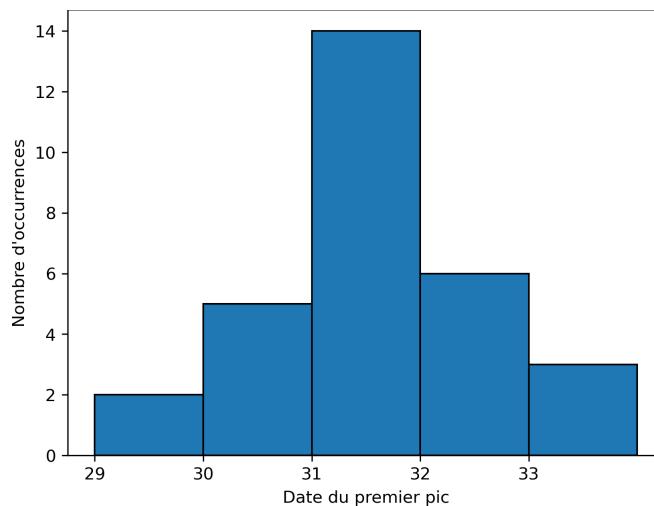
Distribution de la date du premier pic  
pour le langage Python



Distribution de la date du premier pic  
pour le langage C



Distribution de la date du premier pic  
pour le langage C++

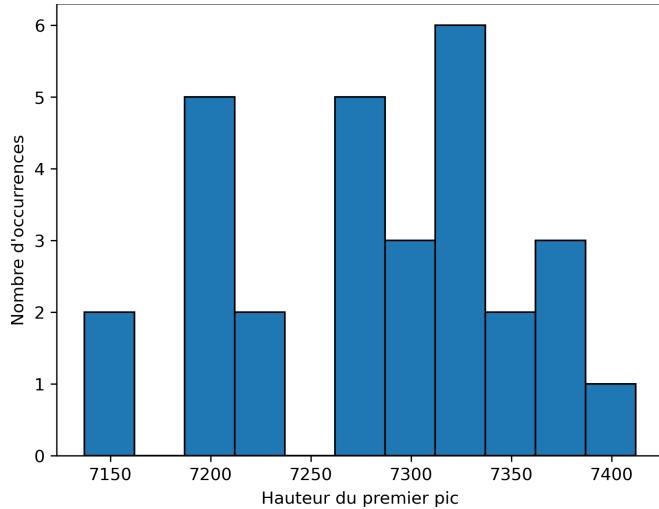


On note ces résultats du nombre d'occurrences de la date du premier pic en fonction du langage :

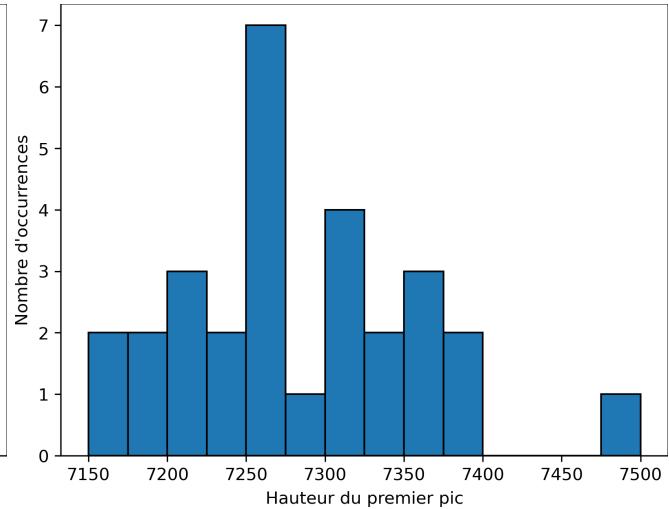
Date du premier pic	Langage		
	Python	C	C++
29	1	1	2
30	7	11	5
31	20	11	14
32	2	6	6
33	0	1	3

On trace ces graphiques (en histogramme) de la distribution de la hauteur du premier pic :

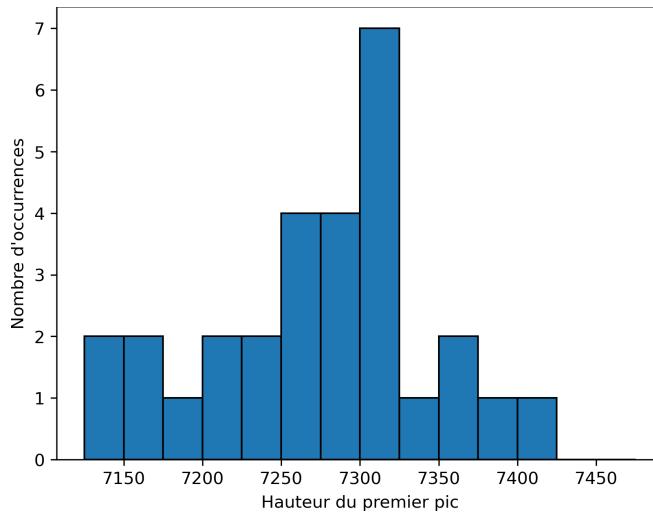
Distribution de la hauteur du premier pic  
pour le langage Python



Distribution de la hauteur du premier pic  
pour le langage C



Distribution de la hauteur du premier pic  
pour le langage C++



On note ces résultats de la valeur  $p$  des tests de Shapiro (arrondis à  $10^{-2}$ ) :

Type de résultat testé	Langage		
	Python	C	C++
Date du premier pic	<0.01	<0.01	0.01
Hauteur du premier pic	0.83	0.24	0.64

On note ces résultats des tests statistiques (arrondis à  $10^{-2}$ ) :

Type de résultat testé	Type de test	Type de résultat		
		Statistique $f$	Statistique $h$	Valeur $p$
Date du premier pic	Kruskal-Wallis	/	2.28	0.32
Hauteur du premier pic	ANOVA	0.07	/	0.94

### Commentaires

- Mathématiquement, la date du premier pic correspond à l'argmax de la fonction  $I$  et la hauteur du premier pic correspond au max de la fonction  $I$ . Pour les obtenir, on utilise ces fonctions avec le tableau I.
- Avant de faire les tests statistiques, on voit que les statistiques (moyennes, extremums, quartiles, écarts-types et distributions) de la date et hauteur du premier pic semblent très proches entre les langages.
- Pour vérifier si les dates et hauteurs du premier pic suivent une distribution normale, on fait un test de Shapiro (pour chaque langage).
- Pour chaque langage, le test de Shapiro montre que la date du premier pic ne suit pas une distribution normale (car les valeurs  $p$  sont inférieure à 0.05), donc le test adapté à faire ensuite est le test de Kruskal-Wallis.
- Pour chaque langage, le test de Shapiro montre que la hauteur du premier pic suit une distribution normale (car les valeurs  $p$  sont supérieures ou égales à 0.05), donc le test adapté à faire ensuite est le test ANOVA.
- Les tests de Kruskal-Wallis et ANOVA montrent bien que les variations de la date et hauteur du premier pic sont plutôt liées à la graine et non au langage (car les valeurs  $p$  sont supérieures ou égales à 0.05).

### 2.2.2.3 Comparaison des temps d'exécution

On note ces temps d'exécution (sur les 30 graines, en s, arrondis à  $10^{-2}$ ) en fonction du langage :

Type de temps d'exécution (en s)	Langage		
	Python	C	C++
Moyenne	76.08	0.19	0.28
Écart-type	2.33	<0.01	<0.01

### Commentaires

- Le meilleur langage est le langage C suivi de près par le langage C++. Le langage Python est très mauvais.
- On note ces résultats d'autres tests :
  - Les temps d'exécution sont d'environ 2.0 s (au lieu de 0.2 s et 0.3 s) pour les langages C et C++ si on n'inline pas les fonctions.
  - Les temps d'exécution sont d'environ 55 s (au lieu de 76 s) pour le langage Python si on utilise la méthode `shuffle` au lieu de l'algorithme de Fisher-Yates dans la fonction `melanger_ordre`.
  - Les temps d'exécution sont d'environ 0.2 s (au lieu de 0.3 s) pour le langage C++ si on utilise la méthode `shuffle` au lieu de l'algorithme de Fisher-Yates dans la fonction `melanger_ordre`.
- On garde tout de même les méthodes un peu moins optimisées pour pouvoir avoir la meilleure comparaison statistique possible entre les langages.
- Les temps d'exécution ne sont pas enregistrées dans des fichiers csv, ils sont affichés dans le terminal pendant les exécutions (voir /Autres/temps\_execution.txt).

### 2.2.3 Autres résultats à explorer

On note quelques autres résultats à explorer :

- Faire une analyse plus poussée de la consommation de la mémoire en fonction des langages.
- Faire une analyse de la consommation d'énergie en fonction des langages.
- Refaire toute l'analyse statistique avec d'autres générateurs de nombres pseudo-aléatoires.