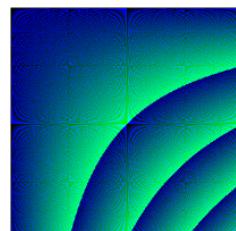


17 janvier 2014

PDG



---

# Propagation Simulator

---

Manuel utilisateur

Groupe 3

Decorvet Grégoire, Froger Hadrien,  
Jaquier Kevin, Schweizer Thomas,  
Sinniger Marcel

**heig-vd**

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

## Table des matières

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                        | <b>1</b>  |
| 1.1      | Aperçu . . . . .                           | 2         |
| <b>2</b> | <b>Structure</b>                           | <b>3</b>  |
| 2.1      | Structure minimale . . . . .               | 3         |
| <b>3</b> | <b>Démarrage</b>                           | <b>4</b>  |
| 3.1      | Configuration . . . . .                    | 4         |
| 3.2      | Remarques . . . . .                        | 4         |
| <b>4</b> | <b>Notions élémentaires</b>                | <b>6</b>  |
| 4.1      | Syntaxe JSON . . . . .                     | 6         |
| 4.2      | Configuration de démarrage . . . . .       | 8         |
| 4.3      | Configuration minimale . . . . .           | 8         |
| 4.4      | Nommage . . . . .                          | 9         |
| 4.4.1    | Noms des simulations . . . . .             | 9         |
| 4.4.2    | Noms des entités et des terrains . . . . . | 9         |
| 4.4.3    | Noms des actions . . . . .                 | 10        |
| <b>5</b> | <b>Création d'une simulation</b>           | <b>11</b> |
| 5.1      | Outils de génération de fichiers . . . . . | 11        |
| 5.2      | Nouvelle simulation . . . . .              | 11        |
| 5.2.1    | Génération des fichiers . . . . .          | 11        |
| 5.3      | Nouvelle entité . . . . .                  | 15        |
| 5.3.1    | Génération des fichiers . . . . .          | 15        |
| 5.3.2    | Paramètres . . . . .                       | 15        |
| 5.4      | Nouvelle action . . . . .                  | 15        |
| 5.4.1    | Génération des fichiers . . . . .          | 15        |
| 5.4.2    | Paramètres . . . . .                       | 16        |
| 5.5      | Nouvelle interaction . . . . .             | 16        |
| 5.5.1    | Génération des fichiers . . . . .          | 16        |
| 5.5.2    | Paramètres . . . . .                       | 16        |
| <b>6</b> | <b>Ecriture de scripts simples</b>         | <b>17</b> |
| <b>7</b> | <b>Ecriture de scripts avancés</b>         | <b>20</b> |

|          |  |           |
|----------|--|-----------|
| 7.1      | Syntaxe complète des fichiers JSON . . . . .     | 20        |
| 7.1.1    | Créer un nouveau trigger . . . . .               | 21        |
| 7.1.2    | Fonction utilisateur . . . . .                   | 21        |
| 7.2      | Emplacement des scripts . . . . .                | 23        |
| 7.2.1    | Script absolu . . . . .                          | 23        |
| 7.2.2    | Script relatif . . . . .                         | 23        |
| 7.2.3    | Les interactions . . . . .                       | 23        |
| <b>8</b> | <b>Export</b>                                    | <b>26</b> |
| 8.1      | Configurer ses exports . . . . .                 | 28        |
| 8.1.1    | Redéfinir l'export par défaut . . . . .          | 28        |
| 8.1.2    | Définir ses propres exports . . . . .            | 28        |
| <b>A</b> | <b>Identifiants de couleurs</b>                  | <b>30</b> |
| <b>B</b> | <b>Exemple de simulation : Recherche de nids</b> | <b>32</b> |
| B.1      | Présentation . . . . .                           | 32        |
| B.2      | Définition des terrains . . . . .                | 33        |
| B.3      | Implémentation . . . . .                         | 33        |
| B.3.1    | Déplacement des fourmis . . . . .                | 34        |
| B.3.2    | Placement des nids . . . . .                     | 40        |
| B.3.3    | Évaluation des nids . . . . .                    | 46        |
| B.3.4    | Tandem run recruteuse - chercheuse . . . . .     | 51        |
| B.3.5    | Terrain mortel . . . . .                         | 53        |
| B.3.6    | Export . . . . .                                 | 54        |
| <b>C</b> | <b>Feuille récapitulative</b>                    | <b>56</b> |

## Chapitre 1

# Introduction

Merci d'avoir acheté *Propagation Simulator* .

Nous sommes heureux de vous présenter ce nouveau simulateur de propagation.

L'objectif principal de *Propagation Simulator* est de simplifier le travail de construction d'un moteur de simulation.

*Propagation Simulator* propose donc de nombreuses voies de configurations pour définir les terrains et les entités de la simulation.

Chaque entité de la simulation peut se déplacer différemment selon les types de terrains, et peut définir de nombreuses propriétés supplémentaires.

La souplesse de l'application permettra rapidement de créer des simulations simples, puis de les complexifier au fur et à mesure.

Toutes les simulations que vous pourrez lancer sont reproductibles, vous pouvez ainsi transmettre vos simulations à d'autres personnes.

## 1.1 Aperçu

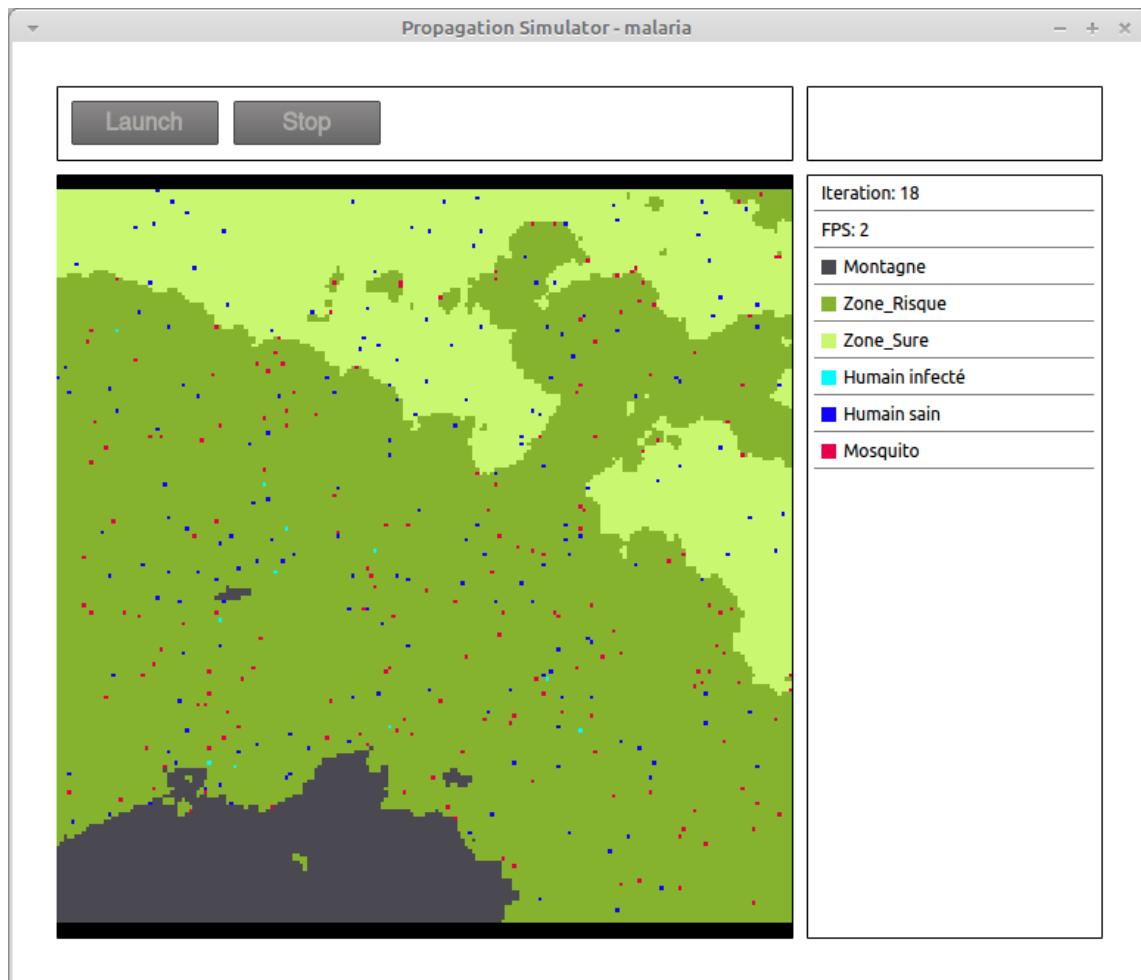


FIGURE 1.1: *Propagation Simulator* en fonction ! Des humains et des moustiques se déplacent sur deux types de terrains. Les moustiques piquent les humains.

## Chapitre 2

# Structure

Le simulateur *Propagation Simulator* permet des configurations très avancées de vos simulations de propagation. Il vous offre pour cela une architecture flexible, qui contient cependant une structure minimale.

### 2.1 Structure minimale

Pour l'exemple, nous avons pris une structure de dossier définissant deux entités. La structure minimale pour les dossiers s'étend bien entendu aux éventuelles nouvelles entités.



FIGURE 2.1: Capture d'écran d'un dossier de simulation avec des configurations minimales

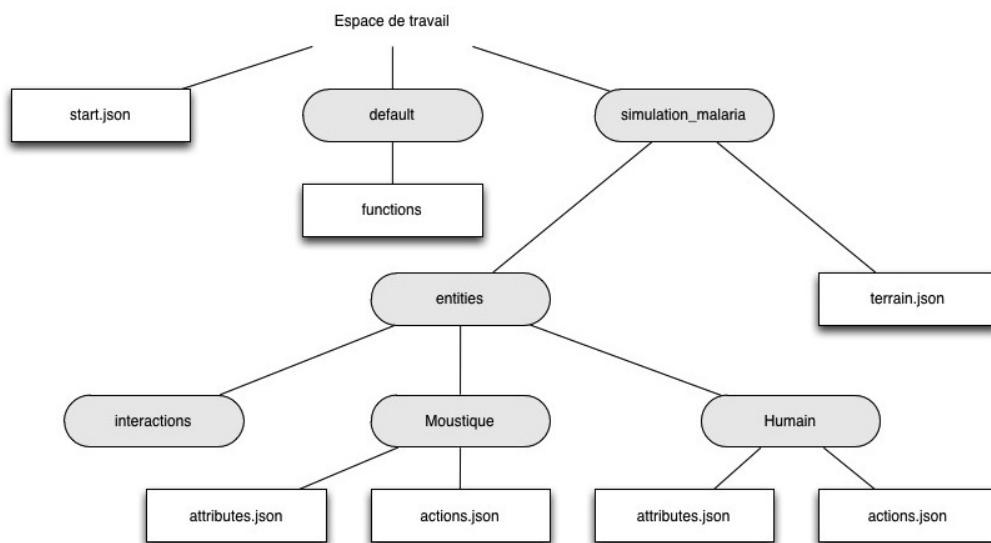


FIGURE 2.2: Schéma de la structure minimale. Les éléments grisés sont des dossiers, les carrés sont des fichiers

## Chapitre 3

# Démarrage

La version installée comporte quelques simulations d'exemple. Afin que vous ayez un bref aperçu du simulateur, nous allons configurer et lancer une de ces simulations.

### 3.1 Configuration

Dans la racine de votre dossier de travail *Propagation Simulator*, vous trouverez un fichier *start.json*. Ouvrez-le avec l'éditeur de votre choix et lisez-le rapidement.

Ce fichier est écrit dans un format nommé JSON<sup>1</sup>. Vous ne devez pour l'instant pas apprendre sa syntaxe, juste comprendre comment le format fonctionne globalement.

Nous avons dans les formats JSON un système de clé-valeur, basé sur cette syntaxe :

Exemple d'un fichier JSON

```
{  
    "cle1" : "valeur1",  
    "cle2" : "valeur2"  
}
```

Le fichier définit le point d'entrée et les paramètres des diverses simulations d'exemples (ainsi que les simulations que vous créerez). Repérez l'en-tête suivant :

```
//  
// MALARIA  
//
```

Il s'agit juste d'un commentaire indiquant que les paramètres qui suivent concernent la simulation *malaria*. Vous pouvez constater que le premier paramètre est donné par la clé *launch* et contient la valeur *malaria*. Cette valeur représente le nom du dossier contenant la simulation.

Au lancement de la simulation, le moteur lira ce fichier et trouvera le nom de la simulation à lancer. C'est avec ce même système que vous pourrez configurer toutes vos simulations.

### 3.2 Remarques

Vous pouvez constater que d'une exécution à l'autre, les simulations se comportent exactement de la même manière que les autres lancements.

---

1. [http://fr.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](http://fr.wikipedia.org/wiki/JavaScript_Object_Notation)

En effet, nous garantissons la reproductibilité avec les clés `seed` et `map_seed` du fichier `start.json`.

Si vous transmettez ces valeurs entières à une personne qui utilise la même simulation que vous, il pourra reproduire à l'identique le fonctionnement de cette simulation et obtenir les mêmes résultats.

## Chapitre 4

# Notions élémentaires

Nous allons parcourir un exemple de configuration afin de comprendre la configuration de l'application.  
Pour cela il est nécessaire de comprendre la syntaxe de base de JSON.

### 4.1 Syntaxe JSON

La syntaxe JSON est de plus en plus utilisée comme format de données, car il est léger et reste très lisible.

Il se base sur un système clé-valeur, comme les annuaires téléphoniques. Par exemple une personne est identifiée par son nom et son prénom.

Ces deux éléments sont considérés comme clé. Dès que vous possédez le nom et le prénom d'une personne, vous connaîtrez la valeur associée à la clé, c'est-à-dire son adresse, numéro de téléphone etc.

Le simulateur *Propagation Simulator* fonctionne sur le principe de l'annuaire. Il connaît certaines clés pré-définies qu'il utilise pour se configurer.

En modifiant les valeurs associées aux clés, vous modifierez le comportement du simulateur.

En JSON, pour écrire un ensemble de clé-valeur, nous utilisons un "dictionnaire". Ce dictionnaire est uniquement une liste de clés et de valeurs.

Par exemple :

Exemple de dictionnaire clé-valeur

```
{  
    "John Doe" : "Chestnut St , Blue Rapids , Kansas" ,  
    "Martine Unetelle" : "Place des Planches 1, 1820 Montreux" ,  
    "Pierre Michu" : "Avenue de Saxe , 75007 , Paris"  
}
```

Mais les valeurs peuvent potentiellement prendre d'autres dictionnaires...

Exemple de valeurs de plusieurs types dans un dictionnaire

```
{  
    "John Doe" : {  
        "Rue" : "Chestnut St" ,  
        "ZIP" : 1200 ,  
        "Ville" : "Kansas City"  
    } ,  
    "Martine Unetelle" : {  
        "Rue" : "Place des Planches" ,  
        "ZIP" : 1820 ,  
        "Ville" : "Montreux"  
    } ,  
    "Pierre Michu" : {  
        "Rue" : "Avenue de Saxe" ,  
        "ZIP" : 75007 ,  
        "Ville" : "Paris"  
    }  
}
```

```

    "Rue" : "Place des Planches 1",
    "ZIP" : 1820,
    "Ville" : "Montreux"
}
"Pierre Michu" : {
    "Rue" : "Avenue de Saxe",
    "ZIP" : 75007,
    "Ville" : "Paris"
}
}

```

Vous pouvez remarquer quand dans l'exemple précédent, les valeurs de ZIP ne sont pas entourées de guillemets.

Les guillemets servent à marqué un champs de texte, si vous ne mettez pas de guillemet il peut s'agir de :

- Un nombre entier. Ex : 42
- Un nombre à virgule flottante (ou nombre réel). Ex : 3.145
- Un booléen (pour les décisions vrai ou faux). Ex : true, false
- Rien, à marquer par le symbole : null

Il existe un dernier type de donnée, les tableaux. Il s'agit d'une suite de valeurs, qui ne sont pas référencées par une clé.

Par exemple, nous aimerais stocker toutes les personnes sous une seule clé.

#### Exemple de liste JSON

```
{
    "Habitants" : [ "John Doe" , "Martine Uonetelle" , "Pierre Michu" ]
}
```

Les crochets symbolisent donc des tableaux. Vous pouvez y intégrer tout type de donnée, comme des dictionnaires ou d'autres tableaux.

#### Exemple de liste JSON avancé

```
{
    "Habitants": [
        {
            "Prenom" : "John",
            "Nom" : "Doe",
            "Animaux Domestiques" : []
        },
        {
            "Prenom" : "Martine",
            "Nom" : "Uonetelle",
            "Animaux Domestiques" : [ "Looping" , "Vanille" , "Pachat" ]
        },
        {
            "Prenom" : "Pierre",
            "Nom" : "Michu",
            "Animaux Domestiques" : [ "Pirate" ]
        }
    ]
}
```

Nous avons donc ici : une liste d'habitants qui contient des dictionnaires. Chaque dictionnaire correspond à une personne et décrit le prénom, nom et la liste de ses animaux domestiques. Ainsi John Doe n'a pas d'animaux, mais Martine Uonetelle en a trois.

Pour finir la description de la syntaxe utilisée dans le simulateur, il nous faut parler des commentaires. La syntaxe JSON n'autorise pas de commentaire dans les fichiers, nous en avons ajouté à la syntaxe. Ils suivent la syntaxe des langages actuellement populaires.

Il existe deux types de commentaire : Des commentaires de lignes, pour décrire quelque chose en une ligne, et les commentaires multi-lignes.

Commentaires dans les fichiers de configuration

```
{  
    "John Doe", // Ceci est un commentaire de ligne  
    "Martine Unetelle",  
    /*  
        Ceci  
        est un  
        commentaire  
        multilignes.  
    */  
    "Pierre Michu"  
}
```

Tout ce qui est commenté sera ignoré par le simulateur, vous pouvez donc expliquer vos clés, afin de reprendre plus tard vos configurations plus simplement.

## 4.2 Configuration de démarrage

Vous avez déjà édité ce fichier lors du premier essai de lancement.

En effet, le fichier *start.json* contient tous les paramètres de lancement de l'application. Vous pouvez modifier les valeurs de ces paramètres à votre guise ou même changer la simulation à lancer.

Comme vous avez déjà pu le remarquer, toutes les simulations d'exemple sont présentes dans le fichier et organisées de façon à ce que les paramètres de chaque simulation se trouvent sous l'en-tête correspondant. Ainsi, pour changer de simulation, plutôt que de modifier la valeur de la clé `launch`, vous pouvez désactiver la simulation courante en mettant ses paramètres en commentaire (en les entourant de `/*` et `*/`), et en activer une autre en lui ôtant ces mêmes symboles.

**Remarque importante :** une seule simulation peut être active à la fois, et chaque paramètre ne doit être défini qu'une seule fois.

La description des paramètres est donnée au chapitre 5.2.1 Démarrage.

## 4.3 Configuration minimale

Afin de commencer la configuration d'une simulation, il est important de connaître les différentes utilités de chacun des fichiers de base.

TABLE 4.1: Description de la structure de fichiers.

| Fichier                                     | Description  |
|---|--|
| /start.json                                 | Démarrage du simulateur  |
| /default                                    | Dossier contenant les scripts par défaut   |
| /default/functions                          | Dossier contenant les fonctions par défaut   |
| /malaria                                    | Dossier de la simulation d'exemple "malaria"   |
| /malaria/terrain.json                       | Les fichiers décrivant les terrains de la simulation   |
| /malaria/entities                           | Tout ce qui concerne les entités.<br>C'est à dire les descriptions des entités et les interactions |
| /malaria/entities/interactions              | Dossier contenant toutes les interactions entre les entités  |
| /malaria/entities/Moustique                 | Dossier contenant les spécifications d'une entité Moustique.<br>Pour l'exemple                     |
| /malaria/entities/Moustique/actions.json    | Description des actions déclenchées lorsque l'entité est sur les terrains.                         |
| /malaria/entities/Moustique/attributes.json | Attributs disponibles pour l'entité.   |
| /logs/malaria/                              | Le dossier contenant tous les résultats de simulation.   |

## 4.4 Nommage

Le simulateur effectue une vérification sur le nom des clés que vous allez utiliser.

Nous allons donc présenter les différentes règles et coutumes pour le nommage de vos clés.

### 4.4.1 Noms des simulations

Le nom de la simulation est le nom de dossier présent dans l'espace de travail.

Il doit donc :

- Être en minuscule.
- Ne pas comporter d'espace.
- Ne pas comporter d'accent.
- Commencer par une lettre.
- Être composé de 3 caractères au moins.
- Être composé de caractères et d'entiers, et éventuellement du séparateur \_ (soulignement).

Vous pouvez par exemple nommer votre dossier de simulation :

- malaria.
- simulation\_malaria.
- simulation\_s01.

### 4.4.2 Noms des entités et des terrains

Le nom des entités est le nom du dossier dans *entities*. Le nom des terrains est la clé du fichier *terrains.json*. Le nom devra :

- Commencer par une lettre majuscule.
- Tous le reste du nom devra être en minuscule.
- Ne pas comporter d'espace.
- Ne pas comporter d'accent.
- Être composé de 3 caractères au moins.
- Être composé de caractères et d'entiers, et éventuellement du séparateur \_ (soulignement).

Il est déconseillé de nommer des entités avec le même nom qu'un terrain.

#### 4.4.3 Noms des actions

Le nom des actions référencées dans le fichier *actions.json* doivent être :

- Être en minuscule.
- Ne pas comporter d'espace.
- Ne pas comporter d'accent.
- Être composé de 3 caractères au moins.
- Être composé de caractères et d'entiers, et éventuellement du séparateur \_ (soulignement).

#### Actions surchargeables

Les actions peuvent surcharger des actions par défaut. Elles sont dans ce cas nommées avec un trait de soulignement (\_).

Les actions surchargées doivent toujours s'exécuter. Il ne faut donc pas définir de fonction `is_performed`

Les noms d'actions que vous pouvez surcharger sont :

TABLE 4.2: Description des actions par défaut surchargeables

| Nom de l'action               | Description  |
|-------------------------------|--|
| <code>_init_entity</code>     | Appelée lors de l'initialisation de l'entité par le simulateur.                                  |
| <code>_create_entities</code> | Appelée une seule fois pour placer les entités sur la carte. Doit retourner une liste d'entités. |

Fichier `actions.json`, pour générer des humains sur la carte à des terrains donnés

```
{
    "_init_entity": {
        "action": [
            // Cree des humains dans les deux zones
            // Ou il peut se deplacer
            [ "Zone_Risque", "Zone_Sure" ]
        ]
    },
    "_create_entities": {
        "action": [
            "creation_instance_num",
            [200] // Cree 200 humains
        ]
    }
}
```

Les subtilités des scripts sont définies plus tard, dans la partie Syntaxe complète des fichiers JSON, page 20.

## Chapitre 5

# Création d'une simulation

## 5.1 Outils de génération de fichiers

*Propagation Simulator* vient avec un outil en ligne de commande pour faciliter la création des fichiers de configuration et scripts utilisateurs, en générant tous les fichiers nécessaires lors de la création de la simulation, et lors de l'ajout d'une entité, action ou interaction à une simulation existante.

L'exécutable de l'utilitaire se trouve à la racine de votre dossier de travail *Propagation Simulator* et se nomme `generate` (ou `Generate` ou `Generate.exe`, selon le système d'exploitation et la version utilisée, dans ce cas-là, remplacez `generate` par la bonne commande dans les explications qui suivent). Pour l'exécuter, ouvrez un terminal dans le dossier et entrez `generate --help`. Ceci affichera les commandes possibles.

Pour obtenir de l'aide sur une commande spécifique, entrez `generate {commande} --help`. Par exemple, pour afficher l'aide de la commande de création d'entité, entrez `generate entity --help`.

**Remarque importante :** si vous demandez à créer un item (simulation, entité, etc.) qui est déjà défini, celui-ci sera écrasé et toutes vos modifications seront perdues !

## 5.2 Nouvelle simulation

### 5.2.1 Génération des fichiers

Pour créer la simulation, ouvrez un terminal à la racine de votre dossier de travail *Propagation Simulator* et exécutuez `generate simulation {nom}`, où `{nom}` est le nom à donner à votre simulation. Par exemple, `generate simulation malaria`. Ceci ajoutera la structure suivante au dossier `/simulations` :

Structure de fichiers d'une simulation

```
{nom}
  terrains.json
  entities
    README.md
    interactions
      README.md
  README.md
```

Le fichier `terrains.json` contient les définitions des différents types de terrains qui seront utilisés pour générer la carte. Le dossier `entities` contiendra les définitions des entités, et le sous-dossier `interactions` contiendra les définitions des interactions entre ces entités. Enfin, les fichiers `README.md` décrivent simplement le contenu du dossier. Vous pouvez les modifier à votre guise pour documenter votre simulation.

## Démarrage

Le fichier *start.json* peut définir les paramètres suivants :

TABLE 5.1: Attributs du fichier *start.json*

| Attribut                     | Type             | Description   |
|------------------------------|------------------|---|
| display_enable               | Booléen          | Active ( <b>true</b> ) ou désactive ( <b>false</b> ) l'affichage graphique                  |
| export                       | Booléen          | Active ( <b>true</b> ) ou désactive ( <b>false</b> ) l'export des données                   |
| interaction_range            | Entier           | Distance entre deux entités à partir de laquelle on considère qu'ils entrent en interaction |
| iterations                   | Entier           | Nombre d'itérations à effectuer   |
| launch                       | Texte            | Le nom de dossier contenant la simulation à lancer  |
| seed                         | Entier           | Nombre entier servant à pouvoir reproduire la simulation                                    |
| map_generation               | Texte            | Méthode de génération de la carte ( <b>perlin</b> , <b>alea</b> ou <b>user</b> )            |
| map_generation_params        | Dictionnaire     | Arguments propres à la méthode de génération de la carte                                    |
| map_seed                     | Entier           | Nombre entier servant à pouvoir reproduire la carte générée                                 |
| map_size                     | (Entier, Entier) | Dimensions de la carte (tableau d'entiers avec (hauteur, largeur))                          |
| min_delay_between_refreshes  | Réel             | Délai minimal (en secondes) avant chaque rafraîchissement de l'écran graphique              |
| nb_iterations_before_refresh | Entier           | Nombre d'itérations à simuler avant de rafraîchir l'écran graphique                         |
| export                       | Booléen          | Si l'export doit s'effectuer  |
| export_param                 | Dictionnaire     | Constantes pour la constructions des fichiers d'export                                      |

## Génération de la carte

Comme mentionné il existe trois modes qui permettent de générer une carte :

- **perlin** : Utilisation d'un bruit de Perlin (nécessite les paramètres `map_generation_params`).
- **alea** : Génération de la carte de manière purement aléatoire.
- **user** : Permet à vous de dessiner la carte (nécessite un fichier `map_generation.json`).

### Configuration de `map_generation_params`

TABLE 5.2: Attributs du paramètre `map_generation_params`

| Attribut      | Type   | Description                                  |
|---------------|--------|--|
| <b>alea</b>   |        |  |
| -             | -      | -  |
| <b>perlin</b> |        |  |
| octaves       | Entier | Nombre de superpositions                     |
| frequency     | Réel   | Nombre de changements par unités de longueur |
| persistance   | Réel   | Niveau de détails                            |
| <b>user</b>   |        |  |
| -             | -      | -  |

### Configuration du fichier `map_generation.json`

Comme mentionné ci-dessus, le mode **user** a besoin d'un fichier `map_generation.json` dans le dossier de la simulation (le dossier qui contient aussi le fichier `terrain.json`).

Le premier pas consiste à définir le terrain par défaut dans la section "default\_terrain".

- "terrain" : "<Un terrain existant de `terrain.json`>"

Ensuite, vous pouvez ajouter des rectangles et des ellipses qui représentent des zones d'un certain type de terrain.

- "type" : "<soit Rectangle soit Ellipse>",
- "terrain" : "<Un terrain existant de `terrain.json`>",
- "dimensions" : "[x, y, largeur, hauteur]"
  - Rectangle : x et y représentent le point en bas à gauche du rectangle
  - Ellipse : x et y représentent le centre de l'ellipse.

Les zones configurées peuvent se chevaucher. Dans ce cas, la dernière zone définie se retrouvera devant les autres.

Exemple de fichier `map_generation.json`

```
{
  "default_terrain" : {
    "terrain" : "Normal"
  },
  "special_areas" : [
    {
      "type" : "Rectangle",
      "terrain" : "Difficile",
      "dimensions" : [5, 5, 50, 50]
    },
    {
      "type" : "Ellipse",
      "terrain" : "Mer",
      "dimensions" : [0, 100, 10, 10]
    }
  ]
}
```

**Terrains**

Le fichier `terrain.json` permet la configuration des différents types de terrains de la carte. Chaque entrée de ce fichier utilise le nom du terrain comme clé et une liste d'attributs comme valeur. Aucun attribut n'est obligatoire donc cette liste peut rester vide. Un exemple :

Exemple de fichier `terrain.json`

```
{
  "Ground" : { }, // aura une couleur choisie par le programme
  "Mountain" : { // aura une couleur brune
    "color": "#6B521C"
  },
  "Sea" : { // aura une couleur bleue
    "color": "blue"
  },
}
```

Les attributs re-définissables pour chaque terrain sont les suivants :

TABLE 5.3: Attributs des terrains

| Attribut | Type  | Description   |
|----------|-------|---|
| color    | Texte | Identifiant de couleur (voir annexe <b>Identifiants de couleurs</b> ) |

## 5.3 Nouvelle entité

Les entités peuvent représenter des humains, des moustiques, des fourmis, etc. En d'autres termes un objet actif au sein de la simulation. Vous pouvez leur assigner des attributs qui pourront changer leur comportement ou leur état, et modifier des attributs prédéfinis tel que la couleur d'affichage, afin de paramétriser la façon dont le simulateur les traite.

### 5.3.1 Génération des fichiers

Pour créer une entité, ouvrez un terminal à la racine de votre dossier de travail *Propagation Simulator* et exécutez `generate entity {nom} {simulation}`, où `{nom}` est le nom à donner à votre entité et `{simulation}` le nom de la simulation dans laquelle ajouter l'entité. Par exemple, `generate entity humain malaria` ajoutera l'entité `humain` à la simulation `malaria`.

La structure suivante sera ajoutée au dossier `/simulations/{simulation}/entities` :

#### Structure de fichiers d'une entité

```
{nom}  
  README.md  
  attributes.json  
  actions.json
```

Le fichier `attributes.json` contient la définition des attributs de l'entité, qui pourront être utilisés pour manipuler l'état de l'entité après une action ou une interaction. Les actions sont définies dans le fichier `actions.json`, tandis que les interactions que l'entité peut avoir avec d'autres entités (ou d'autres individus de la même entité) sont définies séparément dans le dossier `/entities/interactions` du dossier de la simulation.

### 5.3.2 Paramètres

La commande de génération d'entité possède également une option `--attr` permettant de spécifier les attributs que possédera l'entité. Les attributs spécifiés seront ajoutés au fichier `attributes.json` et auront la valeur par défaut `" "`. Par exemple, `generate entity humain malaria --attr age taille` créera une entité `humain` dans la simulation `malaria`, avec les attributs `age` et `taille`.

## 5.4 Nouvelle action

Une action définit un comportement spécifique d'une entité. Il s'agit d'un changement d'état spontané, qui n'agit que sur l'entité elle-même, et peut dépendre du type de terrain sur lequel l'entité se trouve, ainsi que de l'état de l'entité.

Une action est définie pour un certain type d'entité. Par exemple, un moustique peut voler, mais pas un humain.

Si plusieurs actions sont définies pour la même entité et modifient les mêmes attributs, le simulateur gère de façon non-déterministe le choix de l'action qui écrasera les modifications des autres, afin de ne pas fausser statistiquement les résultats si plusieurs actions contradictoires sont définies.

### 5.4.1 Génération des fichiers

Pour ajouter une action à une entité, ouvrez un terminal à la racine de votre dossier de travail *Propagation Simulator* et exécutez `generate action {nom} {simulation} {entité}`, où `{nom}` est le nom de l'action, `{entité}` le nom de l'entité et `{simulation}` le nom de la simulation. Par

exemple, `generate action guerir malaria humain` ajoutera l'action `guerir` à l'entité `humain` dans la simulation `malaria`.

Il y a deux variantes pour définir un action : la première est d'en définir les instructions directement dans sa déclaration JSON, la seconde est de créer un script Python pour l'action, pour une plus grande flexibilité. Pour obtenir la seconde variante, il suffit d'ajouter l'option `--script`.

L'exécution de la commande `generate action` ajoute la définition de l'action au fichier `/simulations/{simulation}/entities/{entité}/actions.json`.

Avec l'option `--script`, un fichier `{name}.py` est ajouté au même niveau, et est référencé par la définition dans `actions.json` comme étant le script de l'action.

#### 5.4.2 Paramètres

La définition de l'action se fait par le biais d'un script personnalisé. Voir chapitres 6 Ecriture de scripts simples et 7 Ecriture de scripts avancés.

### 5.5 Nouvelle interaction

Les interactions sont similaires aux actions mais concernent deux entités. Leur déclenchement n'est pas spontané mais survient lorsque deux entités se trouvent suffisamment proches l'une de l'autre (la distance considérée est paramétrable dans le fichier `start.json`).

Une interaction entre deux entités peut dépendre de l'état de chacune des deux entités, et modifier ces états une fois déclenchée. Les interactions sont définies pour deux types d'entités, qu'ils soient distincts ou non. Par exemple, un moustique peut piquer un humain, et un humain peut contaminer un autre humain.

Les interactions sont exécutées une fois que l'application de toutes les actions est terminée. Si plusieurs interactions modifient le même attribut d'une entité, les règles d'applications sont les mêmes que pour les actions.

#### 5.5.1 Génération des fichiers

Pour ajouter une interaction entre deux entités, ouvrez un terminal à la racine de votre dossier de travail *Propagation Simulator* et exécutez `generate interaction {nom} {simulation} {entité1} {entité2}`, où `{nom}` est le nom de l'interaction, `{entité1}` et `{entité2}` le nom des entités concernées et `{simulation}` le nom de la simulation. Par exemple, `generate interaction piquer malaria moustique humain` ajoutera l'interaction `piquer` entre les entités `humain` et `moustique` dans la simulation `malaria`.

Comme pour les actions, les interactions peuvent être définies uniquement par une déclaration JSON, ou par un script Python. Pour créer un script Python, il suffit d'ajouter l'option `--script`.

S'il n'y a que la définition JSON, seul un fichier `{name}.json` est ajouté au dossier `/simulations/{simulation}/entities/{interactions}`.

S'il y a en plus un script Python, le fichier `{name}.py` est ajouté au même niveau.

#### 5.5.2 Paramètres

La définition de l'interaction se fait par le biais d'un script personnalisé. Voir chapitres 6 Ecriture de scripts simples et 7 Ecriture de scripts avancés.

## Chapitre 6

# Ecriture de scripts simples

Afin d'apprendre à écrire des scripts simples, nous allons parcourir un exemple simple.

Le but de l'exemple est que l'humain vieillisse d'une année tous les cent pas de simulation.

Dans l'état, la simulation d'exemple est composée de deux entités : **Humain** et **Moustique**

TABLE 6.1: Entité Humain

| Attributs    |  |
|--------------|--|
| _name        | "Humain"   |
| _plural_name | "Humains"  |
| _color       | "#1200FF"  |
| _labels      | Les labels pour la simulation. Définition de la couleur de chaque label. |
| Actions      |  |
| normal_move  | <b>Terrains</b> : Zone_Risque et Zone_Sure. <b>Script</b> : mvt_entity   |

TABLE 6.2: Entité Moustique

| Attributs     |  |
|---------------|--|
| singular_name | "Moustique"  |
| plural_name   | "Moustiques"   |
| Actions       |  |
| normal_move   | <b>Terrains</b> : Zone_Risque. <b>Script</b> : mvt_moustique |

Les attributs des entités sont définis dans leur fichier *attributes.json*. Pour l'instant, le fichier contient :

Fichier **attributes.json** de Humain

```
{
    "_name" : "Humain",
    "_plural_name" : "Humains",
}
```

Pour implémenter notre script, nous avons deux nouveaux attributs à ajouter. Le nombre de pas de l'humain, et son âge.

Nous allons donc ajouter ces deux attributs avec leurs valeurs de départ. À noter que tous les humains débuteront leur simulation avec cette même valeur. Vous apprendrez plus tard comment faire varier la valeur de départ.

Fichier **attributes.json** de Humain avec les deux attributs supplémentaires.

```
{
    "_name" : "Humain",
    "_plural_name" : "Humains",
    "nombre_pas" : 0,      // Le nombre de pas parcourus
    "age" : 1              // Tous les humains commencent par avoir 1an.
}
```

Nous devons maintenant définir deux actions pour les humains :

1. L'incrément du nombre de pas.
2. Le vieillissement par rapport au nombre de pas effectués.

Les actions sont toutes appelées une par une à chaque pas de simulation.

Elles sont appliquées selon une condition qui leur est propre nommée **is\_performed**.

Les modifications que les actions apportent à l'entité seront appliquées à la fin de chaque pas de simulation.

Dans notre cas, il nous faut incrémenter le nombre de pas tout le temps. Nous allons donc ajouter l'action **increment\_pas**

Fichier **actions.json** de Humain avec une action d'incrément

```
{
    "normal_move": {
        "with" : ["Zone_Risque", "Zone_Sure"],
        "action" : "/function_pack/mvt_entity"
    },
    "increment_pas" : {
        "with" : ["*"], // Tous les types de terrains
        "action" : [
            ["True"], // Le is_performed est toujours vrai
            // increment du nombre de pas de l'humain.
            ["humain.nombre_pas += 1"]
        ]
    }
}
```

Le champs **action** de **increment\_pas** est défini par un tableau de deux éléments. Le premier définit le **is\_performed**, le deuxième définit le **action\_performed**.

Dans la manière la plus synthétique d'écrire des scripts, vous pouvez entrer en **is\_performed** une ou plusieurs conditions qui seront évaluées avec un ET logique. Pour le **action\_performed**, chaque élément du tableau sera considéré comme un script Python et sera exécuté.

Dans notre cas, nous utilisons uniquement les opérateurs de base de Python<sup>1</sup>.

Nous avons donc maintenant des humains qui augmentent leur nombre de pas au fil de la simulation.

Nous allons à présent faire vieillir les humains tous les cent pas par l'action *vieillissement*.

Fichier **actions.json** de Humain avec une action de vieillissement

```
{
    "normal_move": {
        "with" : ["Zone_Risque", "Zone_Sure"],
        "action" : "/function_pack/mvt_entity"
    },
}
```

1. [http://fr.wikibooks.org/wiki/Programmation\\_Python/Op%C3%A9rateur](http://fr.wikibooks.org/wiki/Programmation_Python/Op%C3%A9rateur)

```

"increment_pas" : {
    "with" : "*",
    "action" : [
        ["True"], // Le is_performed est toujours vrai

        // increment du nombre de pas de l'humain.
        ["humain.nombre_pas += 1"]
    ]
},
"vieillissement" : {
    "with" : "*",
    "action" : [
        // Si le nombre de pas a depasse 100
        ["humain.nombre_pas > 100"],

        // Vieillir
        [
            "humain.age += 1", // L'humain vieilli
            "humain.nombre_pas = 0" // Reset le nombre de pas.
        ]
    ]
}
}

```

L'humain va donc ici vieillir tous les cent pas.

A chaque vieillissement le compteur de pas redémarre, permettant ainsi de ré-évaluer la condition 100 pas plus tard.

Même si l'exemple ne sert à rien en l'état, il peut permettre d'ajouter assez rapidement des fonctionnalités supplémentaires à la simulation. Par exemple, si l'humain a dépassé un certain âge, il sera infecté par la malaria plus rapidement.

## Chapitre 7

# Ecriture de scripts avancés

## 7.1 Syntaxe complète des fichiers JSON

Nous allons dans cette partie étudier comment définir des scripts de manière complète. Cela permettra de créer des simulations plus compliquées, et surtout de définir des scripts réutilisables entre les différentes entités.

Étudions tout d'abord un des scripts résultant de la partie précédente.

L'objectif était d'incrémenter un nombre de pas à chaque étape de la simulation.

Fichier `actions.json` de Humain avec une action d'incrément de pas.

```
{  
    "normal_move": {  
        "with" : ["Zone_Risque", "Zone_Sure"] ,  
        "action" : "/function_pack/mvt_entity"  
    },  
    "increment_pas" : {  
        "with" : ["Mountain", "Earth"] ,  
        "action" : [  
            "True"], // Le is_performed est toujours vrai  
  
            // increment du nombre de pas de l'humain.  
            ["humain.nombre_pas += 1"]  
    }  
}
```

Comme nous l'avions dit, une action est définie en deux parties : `is_performed` et `action_performed`. Pourtant, dans le fichier original, une action était définie par une chaîne de caractères, montrant une syntaxe au final plus courte.

Les actions définies par des chaînes de caractères désignent un appel de `Trigger`. Un `Trigger` est une classe Python que vous pouvez écrire dans un fichier Python dédié.

Ces scripts sont dans ce cas définis dans le dossier de l'entité, à côté des fichiers JSON.

L'action nommée `mvt_entity` fait ici référence au fichier Python `mvt_entity.py` qui contient le Trigger `MvtEntity`. Sans surprise, un `Trigger` est défini par deux méthodes statiques `is_performed` et `action_performed`. La méthode `is_performed` doit retourner un booléen et `action_performed` est une procédure n'ayant aucune valeur de retour.

### 7.1.1 Créer un nouveau trigger

Afin de créer un nouveau **Trigger** pour une entité, vous devez créer un fichier Python correspondant au nom du trigger.

Un trigger *foo\_bar.py* devra contenir une classe nommée **FooBar**. Cette classe devra hériter de la classe **Trigger**, du module **simutils**.

Ainsi un trigger vide est défini ainsi :

Fichier *trigger\_vide.py*

```
# -*- coding: utf8 -*-
from simutils import *

class TriggerVide(Trigger):
    @staticmethod
    def is_performed(data):
        """ Toujours appliquer le trigger. """
        return True

    @staticmethod
    def action_performed(data):
        """ Ne rien faire """
        pass
```

Il y a plusieurs choses importantes à noter de l'exemple :

1. Le fichier doit déclarer l'encodage UTF-8, et être effectivement encodé en UTF-8.
2. Le fichier doit importer les outils de simulation avec `from simutils import *`
3. La classe définit deux méthodes statiques, prenant un paramètre `data`.
4. La classe hérite de `Trigger`

Le paramètre `data` vous donne accès aux entités concernées par un **Trigger**.

Un paquet `data` contient les attributs suivants :

TABLE 7.1: Contenu d'un paquet `data` pour un trigger Humain

|                       |  |
|-----------------------|--|
| <code>entity1</code>  | l'entité concernée par l'action, ou l'interaction                        |
| <code>humain</code>   | alias de <code>entity1</code>  |
| <code>humain1</code>  | alias de <code>entity1</code>  |
| <code>entity2</code>  | toujours None pour les actions. 2e entité concernée par une interaction. |
| <code>terrains</code> | liste des terrains où l'action (ou l'interaction) est applicable.        |

Les alias dépendent du nom de l'entité. L'alias est la version minuscule du nom de dossier donné au dossier de l'entité. `entity1` et `entity2` ne sont pas inutiles, nous le verrons lors de la création de Trigger génériques.

### 7.1.2 Fonction utilisateur

Il est possible que vous désiriez créer des **Trigger** qui ont tous le même comportement (`action_performed`), mais qui ne sont pas déclenchés selon la même condition.

Pour résoudre ce problème de copier-coller, nous avons ajouté la notion de **Fonction**. Un **Trigger** n'est qu'une composition de deux **Fonctions** `is_performed` et `action_performed`. Dans vos scripts JSON, vous serez donc capable d'appeler deux fonctions au lieu d'un trigger. Nous l'avons fait de manière implicite lors du premier exemple du vieillissement.

En effet, reprenons uniquement l'action `vieillissement` :

Action *vieillissement* du fichier `actions.json` de Humain

```
"vieillissement" : {
    "with" : ["Zone_Risque", "Zone_Sure", "Montagne"] ,
    "action" : [
        // Si le nombre de pas a depasse 100
        ["humain.nombre_pas > 100"] ,
        // Vieillir
        [
            "humain.age += 1" ,           // L'humain vieilli
            "humain.nombre_pas = 0" // Reset le nombre de pas .
        ]
    ]
}
```

Elle utilise en fait des fonctions par défaut : `is_true` et `exec`, définies dans le dossier `/default/fonctions`. Ce sucre syntaxique permet d'appeler rapidement ces deux fonctions uniquement. Un appel complet serait :

Action *vieillissement* du fichier `actions.json` de Humain, syntaxe complète

```
"vieillissement" : {
    "with" : ["Zone_Risque", "Zone_Sure", "Montagne"] ,
    "action" : [
        [
            "is_true" ,
            ["humain.nombre_pas > 100"]
        ] ,
        [
            "exec" ,
            [
                "humain.age += 1" ,
                "humain.nombre_pas = 0"
            ]
        ]
    ]
}
```

Ces noms de fonctions suivent les mêmes règles que les triggers, vous pouvez donc définir vos propres fonctions directement dans le dossier de l'entité, et ainsi créer une nouvelle fonction.

En reprenant notre exemple, nous souhaiterions déplacer l'incrément de l'âge dans une fonction dédiée, nommée *vieillir*. Nous modifions donc notre action *vieillissement* :

Action *vieillissement* du fichier `actions.json` de Humain, avec appel de fonction

```
"vieillissement" : {
    "with" : ["Zone_Risque", "Zone_Sure", "Montagne"] ,
    "action" : [
        [
            "is_true" ,
            ["humain.nombre_pas > 100"]
        ] ,
        ["vieillir"]
    ]
}
```

Nous décidons donc d'appeler la fonction *vieillir* tous les 100 pas, sans passer de paramètres.

Le fichier de fonction se définit ainsi :

Fonction `vieillissement` dans le dossier Humain

```
# -*- coding: utf8 -*-
from simutils import *

class Vieillissement(Function):

    @staticmethod
    def call(data):
        data.humain.age += 1
        data.humain.nombre_pas = 0
```

Ainsi, certaines parties du code JSON sont déportées dans une fonction dédiée. À noter que la méthode statique `call` peut avoir plus de paramètres.

## 7.2 Emplacement des scripts

Les fichiers `Trigger` et les fichiers de `Function` sont nommés dans le fichier `actions.json`. Le nom donné par le fichier déterminera quel fichier considérer.

### 7.2.1 Script absolu

Tout d'abord, le nom des scripts peut-être absolu. Ainsi l'action `normal_move` de l'exemple peut être écrite ainsi :

Définir des actions par lien absolu

```
"normal_move": {
    "with" : ["Zone_Risque", "Zone_Sure"],
    "action" : "/function_pack/mvt_entity"
}
```

Cette méthode ne dépend pas du système d'exploitation sur lequel vous exécutez la simulation. La racine est celle de votre dossier utilisateur, puis vous pouvez définir le chemin de votre script, sans le `.py`.

Les avantages de cette méthode sont évidents pour des fichiers qui sont utilisés par plusieurs entités : vous avez un comportement commun. Une modification du script modifie toutes les entités. C'est aussi potentiellement plus dangereux, si vous ne savez pas quelle entité utilise le script. De plus cette méthode rend impossible l'utilisation d'alias pour se référer aux entités. Les attributs `entity1` et `entity2` devront systématiquement être utilisés pour lier un script de manière fiable.

### 7.2.2 Script relatif

Si vous indiquez uniquement le nom de script par le nom du script Python que vous souhaitez appeler, il y a deux possibilités :

1. Il existe dans le dossier de l'entité.
2. Il existe dans le dossier par défaut (`/default` pour les `Trigger` et `/default/functions` pour les `Function`).

### 7.2.3 Les interactions

Les interactions permettent de définir une opération entre deux entités. Un moustique qui pique un humain sera représenté par une interaction entre une entité humain et une entité moustique. Il s'agit donc d'un script qui est appelé lorsque deux entités sont en contact, et il agit exactement comme une action. Nous avons donc une fonction `is_performed` permettant de savoir si lors du contact

l'interaction à lieu, et une autre fonction `action_performed` pour permettre de définir les événements et les opérations qui se produisent alors.

Les scripts JSON des interactions ne sont pas dans les dossiers des entités, mais au même niveau que les entités, car une interaction concerne un couple d'entités, il est donc logique que les interactions soient dans un emplacement dédié.

Vous trouverez les interactions dans le dossier `./entities/interactions`, ce dossier contiendra **un fichier par interaction**. Les interactions n'ont pas de nom particulier, étant donné qu'ils sont entièrement gérés par le simulateur. Il est cependant conseillé de nommer le fichier judicieusement pour vous y retrouver. L'interaction "moustique pique humain" pourrait être appelée "piquer.json" par exemple. Voici un extrait de ce fichier :

```
interaction piquer dans le dossier entities/interactions
{
    "entity1" : "Humain",
    "entity2" : "Moustique",
    "interaction" : "piquer"
}
```

Comme vous pouvez le voir la syntaxe est très simple, il faut définir les deux entités concernées par l'interaction, puis le script. Le script est chargé à une exception près comme le script des actions. Vous pouvez donc écrire :

```
interaction piquer dans le dossier entities/interactions
{
    "entity1" : "Humain",
    "entity2" : "Moustique",
    "interaction" : [
        // is_performed
        ["moustique.est_infecte"],

        // action_performed
        ["humain.est_infecte = True"]
    ]
}
```

Vous pourriez aussi créer un **Trigger** permettant de définir complètement cette interaction :

```
interaction piquer.py dans le dossier entities/interactions
# -*- coding: utf8 -*-
from simutils import *

class Piquer(Trigger):
    @staticmethod
    def is_performed(data):
        return data.moustique.est_infecte

    @staticmethod
    def action_performed(data):
        data.humain.est_infecte = True
```

Dans le cas où les interactions désignent des types d'entité différents, le paquet data contient :

TABLE 7.2: Contenu d'un paquet data pour le trigger Humain-Moustique

|           |                       |
|-----------|-----------------------|
| entity1   | l'humain concerné     |
| humain    | alias de entity1      |
| humain1   | alias de entity1      |
| entity2   | le moustique concerné |
| moustique | alias de entity2      |

Dans le cas d'une interaction entre deux instances du même type d'entité, le paquet **data** est comme suit :

TABLE 7.3: Contenu d'un paquet data pour un trigger Humain-Humain

|         |                             |
|---------|-----------------------------|
| entity1 | le premier humain concerné  |
| humain  | alias de entity1            |
| humain1 | alias de entity1            |
| entity2 | le deuxième humain concerné |
| humain2 | alias de entity2            |

## Chapitre 8

# Export

L'export de données représente la partie la plus importante de votre simulation : il s'agit d'extraire les résultats de cette dernière. Ainsi, tous les résultats de simulation sont sauvegardés dans un dossier `/logs` dédié à la racine de votre dossier utilisateur. Dans ce dossier, vous trouverez vos résultats groupés par simulation, puis par date.

Par défaut, une simulation n'exporte aucune donnée. L'export doit en effet se réaliser uniquement une fois que vous avez terminé la configuration de votre simulation. Un export par défaut existe, afin de rendre instantané la génération de résultats.

Tous les exports se font en format `csv`. Il s'agit du format le plus courant et le plus intuitif pour l'export de données. Chaque ligne correspond à un ensemble de données à une certaine itération. Tous les éléments de la ligne sont séparés par un caractère.

Par exemple, un fichier d'export après trois itérations de la simulation de malaria pourrait ressembler à :

Un fichier Humain.csv

```
_iterations;_x;_y;infecte
0;0;0;-
1;2;3;-
2;3;3;x
```

Afin d'activer l'export et tester cette génération de données, vous pouvez modifier le fichier *start.json* ainsi :

```
fichier start.json

"launch" : "fourmis_tandem",
"map_generation" : "alea",
"map_size" : [100, 100],
"iterations" : 500,
"nb_iterations_before_refresh" : 1,
"min_delay_between_refreshes" : 0.5,
"interaction_range" : 0,
"display_enable" : true,

"export" : true,
// Parametres par defaut, a modifier selon vos preferences.
"export_param" : {
    "separator" : ";",
    "new_line" : "\n",
    "extension" : "csv"
}
```

Le comportement par défaut des exports est le suivant : à chaque itération, tous les attributs de chaque entité présente dans la simulation sont écrites dans un fichier d'export portant le nom du type d'entité. De plus à chaque nouvel export, un fichier *Parameters.csv* est créé afin d'y sauver tous vos paramètres de lancement de simulation. Il est important de conserver ce fichier pour des raisons de reproductibilité (les différentes valeurs d'initialisation pour les générateurs aléatoires y sont sauvées).

## 8.1 Configurer ses exports

### 8.1.1 Redéfinir l'export par défaut

Vous avez plusieurs outils afin d'exporter vos données. Dans un premier temps, vous pouvez modifier le comportement par défaut, en surchargeant l'action d'export appelée sur vos entités (nommée `_log`). Exemple d'export personnalisé :

```
fichier Fourmis/actions.json

/** Vos autres actions ici .... */
"_export": {
    "action": [
        "fourmis._log([ '_x', '_y', 'etat', 'best_nest_id'])"
    ]
}
```

Comme vous le voyez, toutes les entités fournissent une méthode `_log` qui permet d'écrire dans le fichier à son nom, uniquement les attributs listés. Attention, la cohérence du fichier est superficiellement vérifiée, il peut arriver que le fichier d'export soit inutilisable si vous faites l'export d'attributs différents aux fil des itérations. Il est à noter que cette méthode `_log` est accessible partout, vous pouvez donc définir un export par défaut sans export, et n'exporter que durant certaines interactions ou actions.

Exemple :

```
fichier Fourmis/actions.json

/** Vos autres actions ici .... */
"_export": {
    "action": [
        "" // Ne fait rien
    ]
}

fichier interactions/evaluation_nids.py

# -*- coding: utf8 -*-
from simutils import *
import math
class EvaluerNids(Function):
    @staticmethod
    def call(data):
        ### VOTRE CODE ICI ####
        data.fourmis._log([ '_x', '_y', 'etat', 'best_nest_id'])
```

### 8.1.2 Définir ses propres exports

Vous pouvez créer vos propres fichiers d'export, afin de créer des exports personnalisés. Afin d'exporter vers un nouveau fichier, vous devez choisir un nom de fichier qui ne soit pas déjà utilisé par le simulateur. Donc, ne nommez pas votre fichier avec le nom des entités, ou **Parameters**, car sinon le simulateur écrira aussi dans votre fichier.

Vous allez ensuite devoir utiliser la méthode d'export proposée par la `SimUtils : write_in_file`. `SimUtils.write_in_file` permet d'ajouter une ligne d'export dans le nom de fichier (sans extension). Le paramètre à fournir pour l'export est un dictionnaire avec en clé le nom de la colonne et en valeur la cellule associée. Le numéro d'itération est systématiquement ajouté en début de chaque ligne. Par exemple, lorsque le moustique pique l'humain, nous aimerais peut être avoir le lieu de la piqûre, le "score" du moustique et le résultat de la piqûre.

Ainsi, nous aurions :

fichier `interactions/piquer.py`

```
# -*- coding: utf8 -*-
from simutils import *
class Piquer(Trigger):
    @staticmethod
    def is_performed(data):
        est_infecte = SimUtils.random().random() < 0.66 \
            and not data.humain._is_affected
        SimUtils.write_in_file("Piqueur", {"piquer.x" : data.humain._x,
                                         "piquer.y" : data.humain._x,
                                         "moustique.score" : data.moustique.infected_count,
                                         "infecte?" : '-' if est_infecte else 'x'})
    return est_infecte

    @staticmethod
    def action_performed(data):
        """ VOTRE CODE ... """
```

fichier généré `/logs/malaria/la date/Piqueur.csv`

```
_iteration;piquer.x;piquer.y;moustique.score;infecte?
0;2;3;0;x
5;50;33;1;-
7;75;25;1;x
20;98;70;2;x
```

Ce fichier sera ensuite utilisable par n'importe quel tableur, ainsi que les logiciels de statistiques. Une utilisation complète des fonctionnalités d'export est décrite dans les annexes B Exemple de simulation : Recherche de nids

## Annexe A

### Identifiants de couleurs

Les couleurs sont représentées par des identifiants textes qui peuvent être exprimés dans l'un des formats suivants :

- #RGB (chaque R, G ou B représente un nombre hexadécimal)
- #RRGGBB
- #RRRGGBBB
- #RRRRGGGGBBBB
- Le nom de la couleur selon les identifiants standards SVG<sup>1</sup> : blue, lightblue, darkblue, lightgreen, black, ...

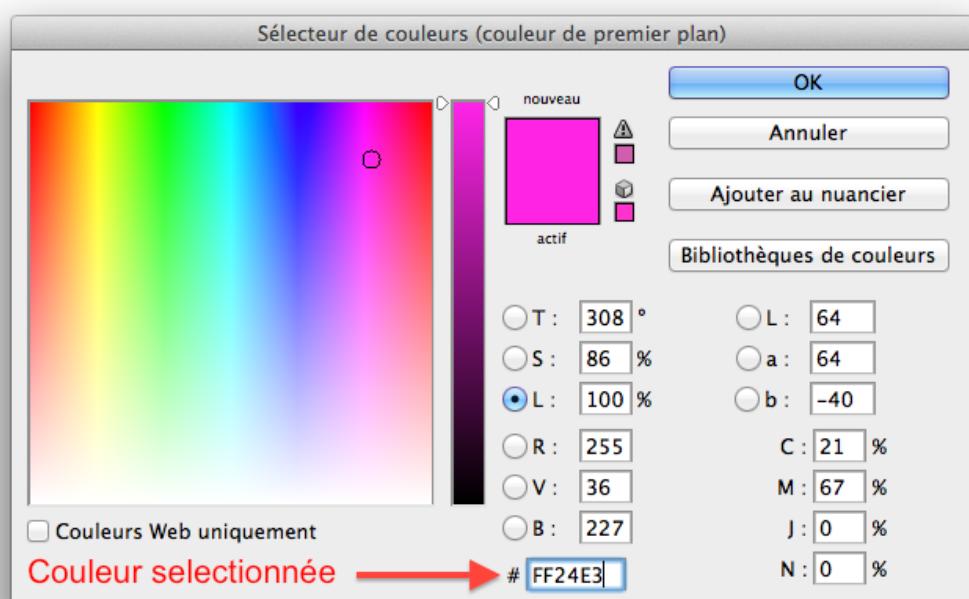
FIGURE A.1: Exemples de couleurs hexadécimales



Afin de choisir vos couleurs, de nombreux outils vous permettent de connaître la valeur hexadécimale d'une couleur. Des outils en ligne, comme <http://colorschemedesigner.com> vous permettront de choisir des couleurs esthétiques. Autrement, la grande majorité des éditeurs d'images comme Photoshop ou Gimp offre cette possibilité.

1. <http://www.w3.org/TR/SVG/types.html#ColorKeywords>

FIGURE A.2: Sélection de couleur par la palette Photoshop



## Annexe B

### Exemple de simulation : Recherche de nids

#### B.1 Présentation

Cet exemple a pour but de créer une simulation de recherche de nids par des fourmis Temnothorax. La situation de départ est que le nid actuel vient d'être détruit, que la reine est tuée, ou que l'environnement ne satisfait plus les conditions de la fourmilière. Des fourmis scouts ou chercheuses, sont chargées de parcourir les zones autour de leur nid afin de découvrir de nouveaux sites potentiels. Chacun de ces nids comportent des avantages physiques, que la chercheuse tâchera au mieux d'évaluer. En plus de la qualité intrinsèque du nid, la fourmi appréciera la distance de son point de départ afin de permettre un déplacement rapide de sa fourmilière.

Une fois qu'une fourmi chercheuse a trouvé un nid, elle l'évalue et si elle le juge suffisamment bon, elle deviendra alors recruteuse. Une fourmi recruteuse a pour objectif de promouvoir son nid auprès des autres fourmis. Elle part donc en se souvenant de la position du nid qu'elle promeut, et amène toutes les chercheuses qu'elle croise, afin qu'elles puissent juger par elles-mêmes la qualité du nid, et ainsi peut-être devenir elle aussi recruteuse.

Ces fourmis sont très particulières. En effet, quand une recruteuse rencontre une chercheuse, la chercheuse se laisse complètement guider par la recruteuse, on appelle ceci le *tandem run*<sup>1</sup>.

Une recruteuse peut aussi rencontrer une autre recruteuse d'un autre nid. L'une d'elle peut alors se laisser persuader et devenir elle aussi recruteuse.

---

1. <http://www.youtube.com/watch?v=T1Hugug4Eik>

## B.2 Définition des terrains

Les terrains sont la base de notre carte, ils définissent les différents types de terrains. Nous allons en premier lieu définir deux types de terrains : **Difficile** et **Normal**. Ces deux types de terrains influeront sur les déplacements des fourmis : un terrain "normal" ne changera rien, alors qu'un terrain "difficile" ralentira les déplacements.

Fichier `/fourmis_tandem/terrain.json`

```
{  
    // Terrain accidenté  
    "Difficile" : { "color": "#75D938" },  
    // Terrain de base  
    "Normal" : { "color": "#C8FAAF" }  
}
```

Nous pouvons maintenant placer nos fourmis afin de tester une première exécution.

## B.3 Implémentation

Pour implémenter la simulation, il est bon de procéder par étape. Ainsi il nous faut créer la simulation en plusieurs étapes, qui testeront si la simulation se comporte correctement. Nous allons donc implémenter :

- Des fourmis qui se déplacent dans la carte.
- Des nids qui se génèrent avec des qualités intrinsèques différentes.
- Des fourmis qui deviennent recruteuses après une évaluation du nid.
- Des recruteuses qui persuadent des chercheuses.
- Des recruteuses qui persuadent des recruteuse.
- Un terrain "Mer", qui peut tuer les fourmis.
- Configuration de l'export des données.

### B.3.1 Déplacement des fourmis

#### Marcher

Afin de simplifier l'implémentation, nous allons créer des fonctions de déplacement paramétrables. Ceci va permettre de définir un paramètre de "vitesse", qui variera selon les différents types de terrains. Afin de créer un déplacement simple, nous avons besoin des attributs suivants :

**direction** Pour déterminer le sens dans lequel se déplace la fourmi

Ainsi, le fichier **attributes.json** du dossier **Fourmis** contiendra simplement :

Fichier **Fourmis/attributes.json**

```
{
    // Direction de deplacement
    //      [G] Gauche
    //      [D] Droite
    //      [H] Haut
    //      [B] Bas
    "direction" : "G"
}
```

À noter qu'il est une bonne pratique de décrire l'ensemble des valeurs possibles au-dessus de l'attribut, afin de ne pas vous perdre plus tard lors de l'implémentation.

Désormais, nous devons implémenter le script qui permettra à la fourmi de se déplacer. Comme dit plus haut, nous allons définir une fonction pour le **action\_performed** uniquement, ce qui permettra de rendre notre code plus modulable.

Fichier **Fourmis/actions.json**

```
{
    "normal_move": {
        "with" : ["Normal"] ,
        "action" :
        [
            [
                ["True"] ,
                [
                    "mvt_fourmis" ,
                    [3]
                ]
            ]
        ],
        "difficile_move": {
            "with" : ["Difficile"] ,
            "action" :
            [
                [
                    ["True"] ,
                    [
                        "mvt_fourmis" ,
                        [1]
                    ]
                ]
            ]
        }
    }
}
```

La fourmi appellera donc `mvt_fourmis` avec une valeur de 1 pour `difficile_move` et une valeur de 3 pour `normal_move`. Ainsi en terrain normal, la fourmi ira trois fois plus vite qu'en terrain accidenté. Il nous faut maintenant implémenter la fonction `mvt_fourmis` qui prend ce paramètre et choisit la position future de la fourmi.

Ce script de déplacement devra donc prendre en compte la vitesse de la fourmi, mais aussi gérer les éventuelles sorties de carte, et les changements de direction spontanés.

## Fichier Fourmis/mvt\_fourmis.py

```
# -*- coding: utf8 -*-
from simutils import *

class MvtFourmis(Function):

    """ Directions possibles d'une fourmi """
    directions = [ 'G', 'D', 'H', 'B']

    @staticmethod
    def call(data, speed):
        direction = data.fourmis.direction
        while True:
            x = data.fourmis._x
            y = data.fourmis._y

            #
            # Calcul de la position future
            # d'après la direction de la fourmi et sa
            # vitesse.
            #
            if direction == 'G' :
                x -= speed
            elif direction == 'D':
                x += speed
            elif direction == 'B':
                y -= speed
            elif direction == 'H':
                y += speed

            # Le point futur est dans la carte, on arrete de chercher.
            if SimUtils.is_in_map_area(x, y):
                break

            # La fourmi bloque, elle doit changer de direction
            # Et retenter avec cette nouvelle direction.
            direction = SimUtils.random() \
                .choice(MvtFourmis.directions)

            # 20 % de chance de changer de direction
            if SimUtils.random().randint(0, 100) > 80:
                direction = SimUtils.random() \
                    .choice(MvtFourmis.directions)

            # Sauvegarde de la nouvelle direction
            data.fourmis.direction = direction
            data.fourmis._move_entity(x, y)
```

Dans ce script, il y a plusieurs choses importantes à noter : l'utilisation de `SimUtils.random()`, de variables temporaires `x`, `y` et `direction` et de variables statiques à la fonction.

#### `SimUtils.random()`

Afin de permettre la reproductibilité, il est essentiel de n'utiliser que le générateur de nombre aléatoire fournit par le simulateur. En effet, l'utilisation de ce générateur de nombre aléatoire permet de garantir la reproductibilité de vos simulations. L'utilisation directe du package `random` fourni par le langage Python est donc **fortement déconseillée**.

#### Application des changements sur une entité

La modification des attributs d'une entité lors de la simulation ne se fera **qu'à la fin du pas de simulation**. Accéder à la valeur d'un attribut d'une entité renverra la valeur de l'attribut à la fin du pas de simulation précédent. Il est donc fortement conseillé d'effectuer de type de modifications dans des variables locales au script, que l'on attribue à l'entité à la fin de l'exécution du script.

#### Variable statique dans une fonction

Cette utilisation permet de créer une constante `directions` regroupant les différentes directions possibles de la fourmi. Cette implémentation est légère car cette liste existe en un seul exemplaire pour tous les appels de la fonction, ce qui est donc beaucoup plus léger que de créer un attribut. À noter cependant que la modification d'attributs statiques modifiera l'attribut pour toutes les entités de la simulation.

Dans cet exemple, nous n'aurions pas pu utiliser uniquement `data.fourmis.direction`, car nous la modifions plusieurs fois afin de trouver une direction qui ne bloque pas la fourmi. Si nous avions utilisé `data.fourmis.direction`, la boucle aurait pu tourner indéfiniment.

#### Tests

Il est conseillé ici de tester vos scripts, régler les éventuelles erreurs de syntaxes et valider le déplacement des fourmis en terrain normal et accidenté.

Vous pouvez aussi ajouter quelques fioritures aux attributs des fourmis, en surchargeant par exemple les attributs `_name`, `_plural_name` ou `_color`.

#### Fichier Fourmis/attributes.json

```
{
    "_name" : "Fourmis",
    "_plural_name" : "Fourmis",
    "_color" : "#78116B",

    // Direction de déplacement
    // [G] Gauche
    // [D] Droite
    // [H] Haut
    // [B] Bas
    "direction" : "G"
}
```

De plus, pour contrôler le nombre de fourmis qui se promènent sur votre carte, vous pouvez surcharger l'action `_create_entities`, en utilisant une des fonction qui vous est fournies par défaut :

#### Fichier Fourmis/actions.json

```
{
    /* Actions déjà écrites */
    "normal_move": { /* votre code */ },
    "difficile_move": { /* votre code */ },
```

```

/**
 * Determine le nombre d'entites creees
 */
"_create_entities": {
    "action": [
        "creation_instance_num",
        [10]
    ]
}
}

```

### Débuter au même point

Comme on le précise dans la présentation du problème, toutes les fourmis partent du même point, le nid où elles habitaient auparavant. Il nous faut donc initialiser la position des fourmis. Pour cela, nous pouvons utiliser l'action surchargeables `_init_entity`.

Fichier `Fourmis/actions.json`

```

{
    /* Actions deja ecrites */
    "normal_move": { /* votre code */ },
    "difficile_move": { /* votre code */ },
    "_create_entities": {/* votre code */ },

    /**
     * Initialisation des fourmis.
     * Les positionnes n'importe ou sur la carte
     */
    "_init_entity": {
        "action": [
            "start_point", []
        ]
    }
}

```

La fonction `start_point.py` sera donc appelée. Il est à rappeler que les actions surchargées **n'ont pas de is\_performed**, on ne peut donc pas définir pour une action surchargée un `Trigger`, mais uniquement une `Fonction`.

## Fichier Fourmis/start\_point.py

```
# -*- coding: utf8 -*-
from simutils import *

class StartPoint(Function):
    """ Les coordonnees du point de depart , definis
    une et une seule fois pour toutes les entites => statique """
    points = list()

    @staticmethod
    def call(data):
        # Si les coordonnees n'ont pas encore ete definies , alors
        # il s'agit de l'initialisation de la premiere fourmi.
        if len(StartPoint.points) == 0:
            (width, height) = SimUtils.get_map_bounds()
            # prends un point dans les limites
            # de la carte.
            x = SimUtils.random().randint(0, width - 1)
            y = SimUtils.random().randint(0, height - 1)
            StartPoint.points = [x, y]

        data.fourmis\
            ._move_entity(StartPoint.points[0], StartPoint.points[1])
```

Cette fois-ci, l'utilisation d'une variable statique à l'action nous permet de ne définir qu'un seul point de départ pour toutes les fourmis, qui est de plus générer aléatoirement.

### B.3.2 Placement des nids

Tout comme le placement des fourmis, le placement des nids demande de surcharger l'action par défaut `_init_entity`. Mais cette fois-ci, nous allons aussi utiliser cette surcharge pour initialiser une qualité propre à chaque nids - sa qualité.

Pour définir une qualité de nid, nous devons tout d'abord éditer le fichier `Nids/attributes.json`

Fichier `Nids/attributes.json`

```
{
    "niveau" : 0,
    /** Intervalle dans lequel doit se trouve le nid */
    "niveau_min" : 1,
    "niveau_max" : 100
}
```

Ainsi, lors de la création tous les nids auront un niveau de 0. Nous allons donc définir dans le fichier `Nids/actions.json` l'action d'initialisation pour créer des nids différents.

Fichier `Nids/actions.json`

```
{
    /**
     * Initialisation des nids.
     * Les positionnes sur la carte
     */
    "_init_entity": {
        "action" : ["start_point", []]
    },

    /**
     * Determine le nombre d'entites creees
     */
    "_create_entities": {
        "action": [
            "creation_instance_num",
            [20]
        ]
    }
}
```

Le fichier est très semblable à celui des fourmis. Nous créons donc un fichier `start_point.py` dans le dossier `Nids` afin d'initialiser nos 20 nids créés.

Fichier `Nids/start_point.py`

```
# -*- coding: utf8 -*-
from simutils import *

class StartPoint(Function):

    @staticmethod
    def call(data):
        (width, height) = SimUtils.get_map_bounds()
        terrains = SimUtils.get_terrain_names()

        x = SimUtils.random().randint(0, width - 1)
        y = SimUtils.random().randint(0, height - 1)
```

```

data.nids.niveau = SimUtils.random() \
    .randint(data.nids.niveau_min, data.nids.niveau_max)

data.nids._move_entity(x, y)

```

Nous avons donc ici des nids de niveaux distincts. Nous pouvons voir plusieurs améliorations possibles à l'implémentation de nos nids :

1. Avoir des nids plus gros
2. Définir un gradient de couleur pour les nids

### Créer des nids 4 fois plus gros

Il est possible de créer des nids plus gros. Pour cela, nous allons en fait créer plusieurs nids, qui s'aggloméreront pour n'en former qu'un seul. Afin de créer cet agglomération, nous devons ajouter un numéro unique d'identification du nid. Lors de l'initialisation, c'est-à-dire une fois que la position du nid est connue, nous pouvons "agrandir" le nid en y ajoutant d'autres nids autour.

Cela nous demande de garder les positions des nids déjà posés, afin d'éviter les chevauchements de nids.

Fichier Nids/attributes.json

```
{
    // L'identifiant du nid, pour que les agglomerats
    // puissent posseder le meme identifiant.
    "nid_id" : 0,

    "niveau" : 0,
    "niveau_min" : 1,
    "niveau_max" : 100
}
```

## Fichier Nids/start\_point.py

```
# -*- coding: utf8 -*-
from simutils import *

class StartPoint(Function):
    """ Le dernier id attribue au nids.
    Garantit l'unicite des identifiants. """
    nid_id = 1

    """ Liste de toutes les coordonnees utilisees par les nids
    afin d'éviter les chevauchements """
    positions = list()

    @staticmethod
    def extends(entity_class, x, y):
        # Nous creons un label contenant les coordonnees
        # Plus simple pour comparer ensuite si la coordonnee
        # a déjà été utilisée
        lab = "{};{}".format(x, y)

        # Si on peut poser l'agglomérat
        if SimUtils.is_in_map_area(x, y) \
            and lab not in StartPoint.positions:
            StartPoint.positions.append(lab)
            new_entity = SimUtils.new_entity(entity_class)
            new_entity.nid_id = StartPoint.nid_id
            new_entity._move_entity(x, y)

    @staticmethod
    def call(data):
        (width, height) = SimUtils.get_map_bounds()

        while True:
            x = SimUtils.random().randint(0, width - 1)
            y = SimUtils.random().randint(0, height - 1)
            lab = "{};{}".format(x, y)
            if lab not in StartPoint.positions:
                StartPoint.positions.append(lab)
                break
            # un nid existe déjà à cette position

        data.nids.niveau = SimUtils.random() \
            .randint(data.nids.niveau_min, data.nids.niveau_max)

        # nous récupérons le nom de l'entité "officielle"
        # afin de permettre la création de nouvelle entité
        # dynamiquement dans la simulation
        name = data.nids.__name__

        # Création des agglomérats
        StartPoint.extends(name, x+1, y)
        StartPoint.extends(name, x-1, y)
        StartPoint.extends(name, x, y+1)
        StartPoint.extends(name, x, y-1)
        StartPoint.extends(name, x+1, y+1)
        StartPoint.extends(name, x-1, y-1)
        StartPoint.extends(name, x+1, y-1)
```

```
StartPoint . extends (name , x - 1 , y + 1)

data . nids . nid _ id = StartPoint . nid _ id
data . nids . _ move _ entity (x , y)
StartPoint . nid _ id += 1
```

## Définir des labels et des couleurs par qualité

Il est possible de définir ses propres labels pour la simulation graphique. Nous pouvons ainsi vérifier plus facilement le fonctionnement de notre application si nous pouvons quantifier la qualité d'un nid visuellement. Nous avons donc choisi 4 niveaux de couleurs, que nous précisons dans le fichier `Nids/attributes.json`. L'attribut `_labels` peut ensuite être surchargé, afin de redéfinir les légendes pour les nids. Cet attribut est un dictionnaire qui prend en clé le texte affiché dans la légende, et en valeur la couleur attribuée au texte.

Fichier `Nids/attributes.json`

```
{
    "_name" : "Nids",
    "_plural_name" : "Nids",
    "_color" : "#8E88FF",
    "color_1" : "#7772D6",
    "color_2" : "#615DAF",
    "color_3" : "#4A4785",

    // Redefinition des legendes des nids
    "_labels" : {
        "Nids *" : "#8E88FF",
        "Nids * *" : "#7772D6",
        "Nids * * *" : "#615DAF",
        "Nids * * * *" : "#4A4785"
    },
    "nid_id" : 0,
    "niveau" : 0,
    "niveau_min" : 1,
    "niveau_max" : 100
}
```

Désormais, nous devons modifier l'initialisation des nids afin d'attribuer des couleurs au nids, et à ses agglomérats.

Fichier `Nids/start_point.py`

```
# -*- coding: utf8 -*-
from simutils import *

class StartPoint(Function):
    nid_id = 1
    positions = list()

    @staticmethod
    def extends(entity_class, color, x, y):
        lab = "{};{}".format(x, y)
        if SimUtils.is_in_map_area(x, y) and lab not in StartPoint.positions:
            StartPoint.positions.append(lab)
            new_entity = SimUtils.new_entity(entity_class)
            new_entity._color = color
            new_entity.nid_id = StartPoint.nid_id
            new_entity._move_entity(x, y)

    @staticmethod
    def call(data):
        (width, height) = SimUtils.get_map_bounds()
```

```

while True:
    x = SimUtils.random().randint(0, width - 1)
    y = SimUtils.random().randint(0, height - 1)
    lab = "{};{}".format(x, y)
    if lab not in StartPoint.positions:
        StartPoint.positions.append(lab)
        break
    niveau = SimUtils.random() \
        .randint(data.nids.niveau_min, data.nids.niveau_max)

    data.nids.niveau = niveau
    col1 = 0.4 * (data.nids.niveau_max - data.nids.niveau_min)
    col2 = 0.6 * (data.nids.niveau_max - data.nids.niveau_min)
    col3 = 0.8 * (data.nids.niveau_max - data.nids.niveau_min)

    color = data.nids._color
    # Attribution de la couleur d'apres le niveau.
    if niveau < col1:
        data.nids._color = color = data.nids.color_1
    elif niveau < col2:
        data.nids._color = color = data.nids.color_2
    elif niveau < col3:
        data.nids._color = color = data.nids.color_3

    name = data.nids.__name__

    StartPoint.extends(name, color, x+1, y)
    StartPoint.extends(name, color, x-1, y)
    StartPoint.extends(name, color, x, y+1)
    StartPoint.extends(name, color, x, y-1)
    StartPoint.extends(name, color, x+1, y+1)
    StartPoint.extends(name, color, x-1, y-1)
    StartPoint.extends(name, color, x+1, y-1)
    StartPoint.extends(name, color, x-1, y+1)

    data.nids.nid_id = StartPoint.nid_id
    data.nids._move_entity(x, y)
    StartPoint.nid_id += 1

```

Encore une fois, il est important cette fois de générer le niveau du nid dans une variable temporaire, qui sera ensuite assignée à l'entité. Car nous allons tester cette valeur générée dans **le même pas de simulation**, nous ne pouvons donc pas utiliser les attributs des entités, qui se mettent à jour en fin de pas.

### Astuce pour afficher les entités statiques

Après les tests, vous aurez sûrement remarqué que les fourmis, une fois passées sur un nid "écrasent" littéralement le nid. Il s'agit juste d'un problème d'affichage, lié à l'implémentation du simulateur. En effet, les nids sont statiques, ils ne changent donc jamais et c'est sur ces changements que le simulateur se base pour repeindre des entités. Le terrain est donc ré-affiché par-dessus suite au passage d'une fourmi. Une solution temporaire est d'ajouter une action au nid qui demande au simulateur de repeindre tous les nids à chaque pas de simulation. Ainsi, dans le fichier `Nids/actions.json` nous avons :

```

Fichier Nids/actions.json
/*
 * Actions deja ecrites */

```

```

"_init_entity": { /* votre code */ },
"_create_entities": { /* votre code */ },

< /**
 * Pour garder le nid visible quoiqu'il arrive
 */
"static": {
  "with": "*",
  "action": [
    ["True"],
    ["nids._need_refresh = True"]
  ]
}
}

```

Ainsi, nous demanderont au simulateur que quoiqu'il arrive, la position à laquelle le nid se trouve soit toujours rafraîchie.

### B.3.3 Évaluation des nids

Nous devons maintenant définir des interactions entre les fourmis et les nids. Nous allons donc déterminer des états de fourmis. En effet, une fourmi en état "recruteuse" ne pourra pas évaluer un nid qu'elle rencontre, car elle fait déjà la promotion d'un nid. De plus, nous avons besoin d'ajouter différents attributs à la fourmi pour qu'elle puisse faire son travail de recruteuse. Elle doit donc connaître les informations sur le nid qu'elle promeut, comme sa qualité et sa position.

Nous devons donc reprendre le fichier d'attribut des fourmis `Fourmis/attributes.json`.

## Fichier Fourmis/attributes.json

```
{
    "_name" : "Fourmis",
    "_plural_name" : "Fourmis",
    "_color" : "#78116B",

    // Direction de deplacement
    // [G] Gauche
    // [D] Droite
    // [H] Haut
    // [B] Bas
    "direction" : "G",

    // Etat
    // [C] Chercheuse
    // Une chercheuse parcourt le monde a la recherche de nids
    // qui pourraient convenir pour la nouvelle colonie.
    // Elle peut etre persuadee par des recruteuses d'aller
    // voir un nid, ou simplement rencontre un nid potentiel
    // qu'il faudra evaluer.
    // [R] Recruteuse
    // Une fourmi est convaincue de la qualite du nid.
    // Elle cherche donc autour d'elle d'autres fourmis
    // pour tacher de les convaincre de venir evaluer le
    // nid.
    // [T] Tandem run
    // La fourmi recruteuse conduit une chercheuse.
    // Elle emmene la chercheuse jusqu'au
    // nid qu'il lui faut evaluer.
    // [?] Suit
    // Chercheuse qui suit une recruteuse.
    "etat" : "C", // Toutes les fourmis commencent chercheuses.

    "best_nest" : 0.0,           // Le nid que la recruteuse promeut
    "best_nest_id" : 0,          // l'identifiant du meilleur noeud.
    "best_nest_x" : 0,           // Position du meilleur noeud.
    "best_nest_y" : 0
}
```

Désormais, les fourmis commencent toutes au début en état chercheuse. Il faut ensuite redéfinir nos différentes actions suivant cet état. Ainsi, les fourmis en état chercheuse et recruteuse effectuent les mouvements de déplacements normaux. Les autres états auront des déplacements particuliers.

Fichier `Fourmis/actions.json`

```
"normal_move": {
    "with" : ["Normal"],
    "action" :
    [
        ["fourmis.etat == 'C' or fourmis.etat == 'R'"],
        [
            "mvt_fourmis",
            [3, []]
        ]
    ]
},
"difficile_move": {
    "with" : ["Difficile"],
    "action" :
    [
        ["fourmis.etat == 'C' or fourmis.etat == 'R'"],
        [
            "mvt_fourmis",
            [1, []]
        ]
    ]
},
"_init_entity": {
    "action" : [
        "start_point", []
    ]
},
"_create_entities": {
    "action": [
        "creation_instance_num",
        [12]
    ]
}
}
```

Désormais, il nous faut ajouter une interaction entre les nids et les fourmis afin que les fourmis chercheuses évaluent les nids, et non pas les fourmis d'un autre état. Nous créons donc un fichier `evaluation_nids.json` permettant définir cette interaction.

Fichier `interactions/evaluation_nids.json`

```
{
    "entity1" : "Fourmis",
    "entity2" : "Nids",
    "interaction" : [
        ["fourmis.etat == 'C'", [
            "evaluer_nids", []
        ]]
    ]
}
```

Nous créons ensuite le fichier `.py` pour définir les effets de l'interaction. Le but est de déterminer si la fourmi est ou non persuadée par la qualité du nid, et qu'elle souhaite commencer à le promouvoir.

## Fichier interactions/evaluation\_nids.py

```
# -*- coding: utf8 -*-
from simutils import *
import math
class EvaluerNids(Function):
    @staticmethod
    def call(data):
        """ Permet de definir si la fourmi devient recruteuse pour le nid
        rencontre.
        Ce choix est en rapport avec la qualite du nid.
        """
        persuadee = SimUtils.random().randint(0, data.nids.niveau_max) < data.
        nids.niveau and not malus
        if persuadee:
            data.fourmis.etat = 'R'
            data.fourmis._color = data.fourmis.color_recruteuse
            # Memorisation du nid qu'elle promeut.
            data.fourmis.best_nest = data.nids.niveau
            data.fourmis.best_nest_id = data.nids.nid_id
            data.fourmis.best_nest_x = data.nids._x
            data.fourmis.best_nest_y = data.nids._y
```

Les fourmis changent donc de couleurs quand elles sont persuadées, et continuent leurs déplacements normaux, sans pour autant évaluer les autres nids.

**Prendre en compte les distances**

Afin de rendre l'évaluation des nids plus correcte, nous devons calculer la distance entre le point de départ de la fourmi et le nid actuel. Pour cela, nous avons besoin d'ajouter deux attributs pour que la fourmi se souvienne de son point de départ. Ce point de départ devra ensuite être initialisé dans l'action `_init_entity`.

## Fichier Fourmis/attributes.json

```
{
    "_name" : "Fourmis",
    "_plural_name" : "Fourmis",
    "_color" : "#78116B",

    "direction" : "G",
    "etat" : "C",

    "best_nest" : 0.0,
    "best_nest_id" : 0,
    "best_nest_x" : 0,
    "best_nest_y" : 0,

    "start_point_x" : 0,
    "start_point_y" : 0
}
```

Nous devons ensuite initialiser ce point de départ.

## Fichier Fourmis/start\_point.py

```
# -*- coding: utf8 -*-
from simutils import *
```

```
class StartPoint(Function):
    points = list()

    @staticmethod
    def call(data, dont_go):
        if len(StartPoint.points) == 0:
            (width, height) = SimUtils.get_map_bounds()
            terrains = SimUtils.get_terrain_names()

            x = SimUtils.random().randint(0, width - 1)
            y = SimUtils.random().randint(0, height - 1)
            StartPoint.points = [x, y]

        data.fourmis.start_point_x = StartPoint.points[0]
        data.fourmis.start_point_y = StartPoint.points[1]
        data.fourmis._move_entity(StartPoint.points[0], StartPoint.points[1])
```

Désormais il est simple de calculer la distance entre le point de départ et le point d'arrivée. La librairie `math` standard de Python offre d'ailleurs une méthode pour faciliter plus encore ce calcul de distance : `hypot`.

#### Fichier `interactions/evaluation_nids.py`

```
# -*- coding: utf8 -*-
from simutils import *
import math
class EvaluerNids(Function):
    @staticmethod
    def call(data):
        """ Permet de définir si la fourmi devient recruteuse pour le nid
        rencontré.
        Ce choix est en rapport avec la distance du point de départ et du nid
        actuel, ainsi que la qualité du nid.
        """
        persuadée = False

        # Calcul de la distance entre la position de la fourmi et son point de
        # départ
        distance = math.hypot(data.fourmis._x - data.fourmis.start_point_x,
                               data.fourmis._y - data.fourmis.start_point_y)
        if distance == 0:
            return # Il s'agit du point de départ

        malus = SimUtils.random().randint(0, 70) < (100 - distance * 2)
        persuadée = SimUtils.random().randint(0, data.nids.niveau_max) < data.
        nids.niveau and not malus
        if persuadée:
            data.fourmis.etat = 'R'
            data.fourmis._color = data.fourmis.color_recruteuse
            data.fourmis.best_nest = data.nids.niveau
            data.fourmis.best_nest_id = data.nids.nid_id
            data.fourmis.best_nest_x = data.nids._x
            data.fourmis.best_nest_y = data.nids._y
```

Ainsi nous créons un malus qui prend en compte la distance. Le tout est parfaitement arbitraire, le but étant juste de mettre un facteur distance dans la prise de choix.

#### B.3.4 Tandem run recruteuse - chercheuse

Nous devons ici créer une interaction entre deux fourmis de types différents. Pour cela, nous définissons donc une interaction `persuasion.json` qui permettra de faire passer une chercheuse en tandem run avec une recruteuse.

Fichier `interactions/persuasion.json`

```
{
    "entity1" : "Fourmis",
    "entity2" : "Fourmis",
    "interaction" : [
        ["(fourmis1.etat == 'R' and fourmis2.etat == 'C') or (fourmis1.etat == 'C' and fourmis2.etat == 'R')"],
        ["persuasion", []]
    ]
}
```

Ainsi lors de la rencontre d'une recruteuse et d'une chercheuse l'interaction sera toujours effectuée. Le problème majeur de cette interaction est que la chercheuse peut autant être la première entité que la deuxième. Nous réglerons donc ce problème en passant par le biais de variables temporaires.

Le deuxième problème qui se pose à nous est le fait que la recruteuse doit guider la chercheuse. Ainsi elle devra avoir accès à cette fourmi pour lui ordonner les déplacements. Cela se fait par l'ajout dans le fichier `Fourmis/attributes.json` de l'attribut `tandem`, initialisé à la valeur `null`.

Fichier `interactions/persuasion.py`

```
# -*- coding: utf8 -*-
from simutils import *
import math
class Persuasion(Function):
    @staticmethod
    def call(data):
        chercheuse = data.fourmis1 if data.fourmis1.etat == 'C' else data.fourmis2
        recruteuse = data.fourmis2 if data.fourmis2.etat == 'R' else data.fourmis1
        chercheuse._color = chercheuse.color_tandem
        chercheuse.etat = "?"
        recruteuse._color = recruteuse.color_tandem

        recruteuse.etat = "T"
        # liens de la chercheuse et recruteuse.
        recruteuse.tandem = chercheuse
```

Comme vous pouvez le voir, le passage par des variables temporaires va copier la référence sur la fourmi. Nous avons donc créer une sorte d'alias local pour accéder à la fourmi. Nous avons donc une variable chercheuse qui désignera tantôt l'entité une, tantôt l'entité deux.

La chercheuse passe ensuite en état passif, pour se laisser guider par la recruteuse qui désormais va mener la chercheuse à son nid favoris. Par défaut, ces deux états "T" et "?" n'ont aucun déplacements. Nous allons donc définir des actions afin qu'elle opère leur travail correctement.

Fichier `Fourmis/actions.json`

```
{
    /* Actions déjà écrites */
    "normal_move": { /* votre code ... */ },
    "difficile_move": { /* votre code ... */ },
```

```

    "_init_entity": { /* votre code ... */ },
    "_create_entities": { /* votre code ... */ },
}

"normal_tandem_run" : {
    "with" : [ "Normal" ],
    "action" : [
        [ "fourmis. etat == 'T'" ],
        [ "tandem_run", [3] ]
    ]
},
"difficult_tandem_run" : {
    "with" : [ "Difficile" ],
    "action" : [
        [ "fourmis. etat == 'T'" ],
        [ "tandem_run", [1] ]
    ]
}
}

```

Ainsi, nous définissons uniquement pour les recruteuses un mouvement dans les terrains normaux et difficile. Elles se déplaceront selon le script `tandem_run` qui pour but de ramener la recruteuse, et la chercheuse dans leur nid.

## Fichier Fourmis/tandem\_run.py

```

# -*- coding: utf8 -*-
from simutils import *

class TandemRun(Function):
    @staticmethod
    def call(data, speed):
        # va dans la direction du nid
        suiveuse = data.fourmis.tandem
        if suiveuse._x == data.fourmis._x and \
           suiveuse._y == data.fourmis._y and \
           data.fourmis._x == data.fourmis.best_nest_x and \
           data.fourmis._y == data.fourmis.best_nest_y:
            # Fin du tandem run
            suiveuse.etat = 'C'
            data.fourmis.etat = '-'
            data.fourmis.tandem = None

        horizontal = 0
        vertical = 0
        vitesse = SimUtils.random().randint(1, speed)
        # calcul du deplacement horizontal souhaite
        if data.fourmis._x > data.fourmis.best_nest_x :
            horizontal = -vitesse
        elif data.fourmis._x < data.fourmis.best_nest_x :
            horizontal = vitesse
        # calcul du deplacement vertical souhaite
        if data.fourmis._y > data.fourmis.best_nest_y :
            vertical = -vitesse
        elif data.fourmis._y < data.fourmis.best_nest_y :
            vertical = vitesse

        if data.fourmis._x != data.fourmis.best_nest_x or \
           data.fourmis._y != data.fourmis.best_nest_y:

```

```

    suiveuse._move_entity(suiveuse._x + horizontal, suiveuse._y +
                           vertical)
    data.fourmis._move_entity(data.fourmis._x + horizontal, data.
                               fourmis._y + vertical)

```

Ainsi la chercheuse et la recruteuse se déplacent lors d'une et une seule action. On permet ainsi à la fourmi recruteuse de posséder la référence sur la fourmi chercheuse. Une fois sur le nid, les états changent et cette action ne sera plus appelée. La recruteuse repartira recrutée et la fourmi se remet en état chercheuse, et découvrira par la même occasion le nid où elle a été déposée.

### B.3.5 Terrain mortel

Afin de créer une exemple complet, nous allons voir comment supprimer des entités de la simulation. Nous allons donc créer un terrain mer, qui tuera les fourmis avec une certaine probabilité.

Fichier `/terrain.json`

```
{
  "Difficile" : { "color": "#75D938" },
  "Normal" : { "color": "#C8FAAF" },
  "Mer" : { "color": "#246cff" }
}
```

En ajoutant un terrain supplémentaire, nous pouvons déjà constater que les fourmis qui marchent dessus se retrouvent paralysées. En effet, tant qu'aucune action n'est définie pour ce terrain, aucune action ne s'exécute. Nous allons donc créer une action de `nage` et de `noyade`. Ainsi la fourmi tentera de s'échapper de la mer, et risque de se noyer.

Fichier `Fourmis/actions.json`

```
{
  /* Actions déjà écrites */
  "normal_move": { /* votre code ... */ },
  "difficile_move": { /* votre code ... */ },
  "_init_entity": { /* votre code ... */ },
  "_create_entities": { /* votre code ... */ },
  "normal_tandem_run": { /* votre code ... */ },
  "difficult_tandem_run": { /* votre code ... */ },

  "noyade": {
    "with": ["Mer"],
    "action": [
      ["random.randint(0, 100) < 10"],
      ["SimUtils.kill_entity(fourmis)"]
    ]
  },
  "nage": {
    "with": ["Mer"],
    "action": [
      ["True"],
      [
        "mvt_fourmis",
        [1, []]
      ]
    ]
  }
}
```

Les fourmis se déplacent donc d'un mouvement difficile dans la mer, et ont 10% de chances de mourir. Une fois mortes, la fourmi ne pourra plus interagir avec aucune simulation.

### B.3.6 Export

Pour que l'export fonctionne, il vous faut tout d'abord définir dans le fichier `start.json` l'attribut `export` à `true`. Ainsi un fichier de log s'écrira dans le dossier de simulation de votre dossier `/logs`. Des paramètres d'export supplémentaires peuvent être ajoutés.

Fichier `start.json`

```
{
    "launch" : "fourmis_tandem",
    "map_seed" : 2,
    "map_generation" : "perlin",           // alea , perlin ou user
    "map_generation_params" : {
        "octaves" : 2,
        "frequency" : 2,
        "persistance" : 2
    },
    "map_size" : [100, 100],
    "iterations" : 500,
    "nb_iterations_before_refresh" : 1,
    "min_delay_between_refreshes" : 0.5,
    "interaction_range" : 0,
    "display_enable" : true,
    "export" : true,
    "export_param" : {
        "separator" : ";",
        "new_line" : "\n",
        "extension" : "csv"
    }
}
```

Dès ce moment, un export par défaut s'exécute et tous les champs des fourmis s'exportent à chaque itération. Nous désirons désormais n'exporter que quelques attributs de la fourmi. Pour cela, nous pouvons surcharger l'action `_export` pour n'exporter que ce que nous désirons.

Fichier `Fourmis/actions.json`

```
{
    /* Actions déjà écrites */
    "normal_move": { /* votre code ... */ },
    "difficult_move": { /* votre code ... */ },
    "normal_tandem_run" : { /* votre code ... */ },
    "difficult_tandem_run" : { /* votre code ... */ },
    "noyade" : { /* votre code ... */ },
    "nage" : { /* votre code ... */ },
    "_init_entity": { /* votre code ... */ },
    "_create_entities": { /* votre code ... */ },

    "_export": {
        "action": [
            "fourmis._log(['_x', '_y', 'etat', 'best_nest_id'])"
        ]
    }
}
```

Nous n'avons donc maintenant plus que les informations pertinentes pour faire des statistiques. Admettons ensuite que nous aimerais par contre tester l'évaluation des nids par les chercheuses. Ainsi lorsqu'une chercheuse évalue un nid, nous aimerais faire un export de la qualité du nid et de la distance avec le point de départ.

Pour cela, nous allons modifier l'interaction d'évaluation pour faire des écritures de logs ponctuelles.

Fichier `interactions/evaluation_nids.py`

```
# -*- coding: utf8 -*-
from simutils import *
import math
class EvaluerNids(Function):
    @staticmethod
    def call(data):
        """ Permet de definir si la fourmi devient recruteuse pour le nid
        rencontre.
        Ce choix est en rapport avec la distance du point de depart et du nid
        actuel , ainsi que la qualite du nid.
        """
        persuadee = False

        # Calcul de la distance entre la position de la fourmi et son point de
        # depart
        distance = math.hypot(data.fourmis._x - data.fourmis.start_point_x,
                              data.fourmis._y - data.fourmis.start_point_y)
        if distance == 0:
            return # Il s'agit du point de depart

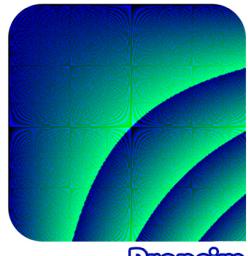
        malus = SimUtils.random().randint(0, 70) < (100 - distance * 2)
        persuadee = SimUtils.random().randint(0, data.nids.niveau_max) < data.
        nids.niveau and not malus

        SimUtils.write_in_file("Evaluation", {"nid.qualite" : data.nids.niveau,
                                              "distance.depart" : distance,
                                              "persuadee" : "X" if persuadee else "-})

        if persuadee:
            data.fourmis.etat = 'R'
            data.fourmis._color = data.fourmis.color_recruteuse
            data.fourmis.best_nest = data.nids.niveau
            data.fourmis.best_nest_id = data.nids.nid_id
            data.fourmis.best_nest_x = data.nids._x
            data.fourmis.best_nest_y = data.nids._y
```

## Annexe C

### Feuille récapitulative



## Methodes

```

String
capitalize()
center(width[, fillchar])
count(sub[, start[, end]])
decode([encoding[, errors]])
encode([encoding[, errors]])
endswith(suffix[, start[, end]])
expandtabs(tabsize)
find(sub[, start[, end]])
format(*args, **kwargs)
index(sub[, start[, end]])
isalnum()
isalpha()
isdigit()
islower()
isspace()
istitle()
isupper()
join(iterable)
ljust(strip[, size])
lstrip(chars)
partition(sep)
replace(old, new[, count])
rfind(sub[, start[, end]])
rindex(sub[, start[, end]])
rjust(width[, fillchar])
rsplit([sep[, maxsplit]])
rstrip(chars)
split([sep[, maxsplit]])
splitlines(keepends)
startswith(prefix[, start[, end]])
strip(chars)
swapcase()
title()
upper())
SimUtils.random()
betavariate(alpha, beta)
choice(seq)
expovariate(lambd)
gammavariate(alpha, beta)
gauss(mu, sigma)
lognormvariate(mu, sigma)
normalvariate(mu, sigma)
paretovariate(alpha)
randint(a, b)
randrange(start[, stop(start,
[step[, int[, default[, maxwidth]]]])])
sample(population, k)
seed([a])
shuffle(x[, random[, int]])
triangular([low[, high[, mode]]])
uniform(a, b)
vonmisesvariate(mu, kappa)
weibullvariate(alpha, beta)

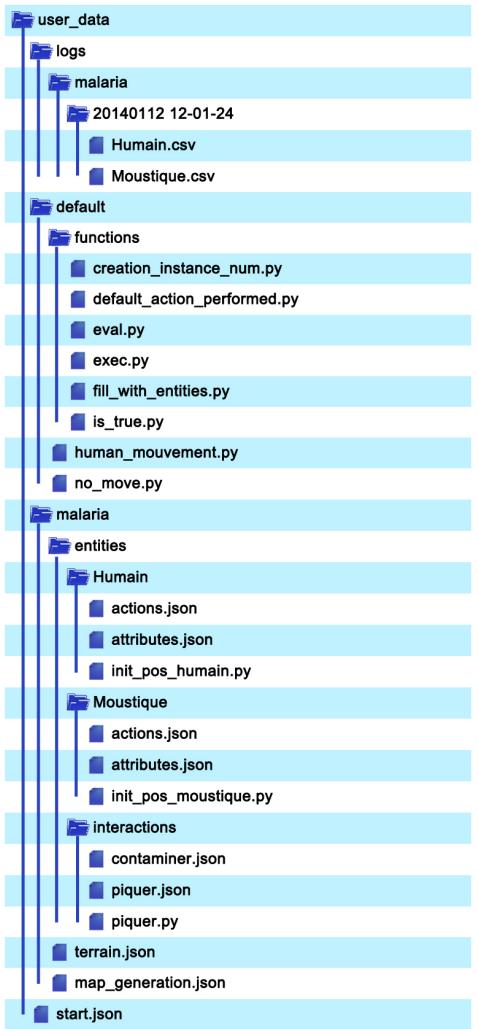
```

```

math
acos(x)
acosh(x)
asin(x)
asinh(x)
atan(x)
atanh(x)
ceil(x)
cos(x)
cosh(x)
degrees(x)
exp(x)
expm1(x)
fabs(x)
factorial(x)
floor(x)
fmod(x, y)
frexp(x)
fsum([iterable])
gamma(x)
hypot(x, y)
isinf(x)
isnan(x)
lgamma(x, i)
log(x[, base])
log10(x)
modf(x)
pow(x, y)
radians(x)
sin(x)
sinh(x)
sqrt(x)
tan(x)
tanh(x)
trunc(x)

```

## Structure



## Actions surchargeables

**init\_entites**  
Appelée lors de l'initialisation des entités. Utile pour placer les entités, ou définir les attributs des entités au départ.  
**Par défaut :** positionne les entités aléatoirement sur la carte.  
**Voir fonctions :** init\_pos\_uniform, init\_entity\_terrains

**create\_entities**  
Donne une liste d'entité au simulateur pour lancer la simulation. Permet de générer le nombre d'entité voulu.  
**Par défaut :** crée 50 entités.  
**Voir fonctions :** creation\_instance\_num

**export**  
Appelée à chaque pas de simulation. Permet de faire des export de données dans des fichiers.  
**Par défaut :** Ecrit tous les attributs de l'entités dans un fichier comportant le nom de l'entité.  
**Voir fonctions :** SimUtils.write\_entity(entity), SimUtils.write\_in\_file(file\_name, object\_dict)

## Fonctions par defaut

**creation\_instance\_num(number)**  
D'après un nombre d'entité désiré, retourne une liste d'entité.

**fill\_with\_entities(rate=1.0)**  
Retourne une liste d'entité. Parcours toute la carte et avec une probabilité "rate" donnée, ajoute une entité à la case donnée.

**Init\_entity\_terrains(terrains\_list)**  
Positionne toutes les entité uniformément sur les terrains donnés.

**init\_pos\_uniform()**  
Initialise les positions des entités aléatoirement sur la carte.

## SimUtils

**SimUtils.random()**  
Retourne un générateur aléatoire initialisé issu du paquetage standard random.

**SimUtils.randcolor(tint=(0,0,0))**  
Donne une couleur aléatoire, avec une teinte de couleur RGB (entre 0 et 255) donné .

**SimUtils.is\_in\_map\_area(x, y)**  
Détermine si la position donnée est dans la carte.

**SimUtils.get\_terrain\_type(x, y)**  
Donne le type de terrain à la position donnée, ou None dans le cas où la position ne se trouve pas dans la carte

**SimUtils.get\_map\_bounds()**  
Retourne un tuple (width, height) contenant la taille de la carte.

**SimUtils.get\_terrain\_names()**  
Retourne tous les noms de terrains utilisés dans la simulation.

**SimUtils.get\_terrains()**  
Retourne les terrains utilisés dans la simulation. Donne accès à la couleur et nom des terrains.

**SimUtils.get\_entities(x, y, when\_out\_of\_bounds=None)**  
Donne les entités à la position donnée, ou when\_out\_of\_bounds quand la position est hors de la carte.

**SimUtils.current\_iteration()**  
Donne le nombre de pas effectué jusqu'à présent.

**SimUtils.new\_entity(entity\_type)**  
Ajoute une nouvelle entité d'après le nom d'entité donné.  
**Voir :** Entity.\_\_name\_\_ S

**SimUtils.write\_in\_file(file\_name, object\_dict)**  
Écrit dans un fichier de logs le dictionnaire donné. La clé du dictionnaire correspond à la colonne. Et la valeur associée la valeur pour la colonne donnée (**Voir fichier** /logs/ma simulation/ma date/)

**SimUtils.write\_entity(entity)**  
Écrit dans un fichier de logs tous les attributs de l'entités donnés.  
**(Voir fichier** /logs/ma simulation/ma date/nom de l'entité)

**SimUtils.kill\_entity(entity)**  
Enlève l'entité de la simulation complètement. Elle n'apparaîtra plus dans les logs.

## Entite

Toutes les entités ont les mêmes méthodes disponibles.

**MonEntite.\_log(attribute\_list)**  
Écrit dans un fichier de log toutes les attributs de l'entité listés. Par défaut, écrit tous les attributs de l'entités sauf \_color, \_labels, \_name et \_plural\_name.

**MonEntite.\_get\_pos()**  
Donne la position de l'entité dans un tuple (x, y)

**MonEntite.\_export()**  
Alias de **MonEntite.\_log( None )**

**MonEntite.\_move\_entity(new\_x, new\_y)**  
Déplace l'entité à la nouvelle position. N'effectue pas le mouvement si il s'agit d'un point en dehors de la carte. Pour la suppression de l'entité, voir **SimUtils.kill\_entity(entity)**

**MonEntite.\_name\_**  
Donne le nom utilisé pour le type d'entité. Utile pour **SimUtils.new\_entity(entity\_type)**

**MonEntite.\_x et MonEntite.\_y**  
Les positions x et y de l'entité.

**MonEntite.\_is\_affected**  
Attribut permettant de savoir si l'entité a été affecté par la propagation. L'action par défaut de is\_performed le met à True.

**MonEntite.\_color**  
La couleur de l'entité dans la simulation.

## Methodes

|   |
|---|
| <b>list</b>                             |
| append(object)                          |
| count(value)                            |
| extend(iterable)                        |
| index(value, [start, [stop]])           |
| insert(index, object)                   |
| pop([index])                            |
| remove(value)                           |
| reverse()                               |
| sort(cmp=None, key=None, reverse=False) |
| <b>dict</b>                             |
| clear()                                 |
| copy()                                  |
| get(k,d)                                |
| has_key(k)                              |
| items()                                 |
| iteritems()                             |
| iterkeys()                              |
| itervalues()                            |
| keys()                                  |
| pop(k,d)                                |
| popitem()                               |
| setdefault(k,d)                         |
| update(E, **F)                          |
| values()                                |
| viewitems()                             |
| viewkeys()                              |
| viewvalues()                            |

**Trigger**  
is\_performed(data, \*args)  
action\_performed(data, \*args)

**Function**  
call(data, \*args)  
execute(data, python\_code)  
eval(data, python\_code)

## Data

Les paquets data sont toujours donnés dans les Trigger et Fonction.  
Il contiennent les entités et les terrains sur lesquels sont les entités.  
Les alias pour les entités dépendent de la situation :

| Situation 1       |  |
|-------------------|--|
| entité 1 : Humain |  |
| entité 2 : None   |  |
| entity1           |  |
| humain            |  |
| humain1           |  |

| Situation 2          |  |
|----------------------|--|
| entité 1 : Humain    |  |
| entité 2 : Moustique |  |
| entity1              |  |
| humain               |  |
| humain1              |  |
| entité 2             |  |
| entity2              |  |
| moustique            |  |
| moustique1           |  |

| Situation 3       |  |
|-------------------|--|
| entité 1 : Humain |  |
| entité 2 : Humain |  |
| entity1           |  |
| humain            |  |
| humain1           |  |
| entité 2          |  |
| entity2           |  |
| humain2           |  |