

|  |
| --- |
| Statistique et Informatique  Projet 1  Bataille Navale |

Monsieur Maxime Bouthors

Philippe MONTEIRO 3800061

Jianwen GAO 28717267

# Table des matières

|  |  |
| --- | --- |
| **Introduction** | 3 |
| **1. Modélisation et fonction simple** | 4 |
| 1.1 Le champ de bataille | 4 |
| 1.2 La fonction placement des bateaux | 4 |
| 1.3 La fonction équivalence des grilles | 4 |
| 1.4 La génération de la grille | 5 |
| 1.5 L’aﬃchage de la grille | 5 |
| **2. Combinatoire du jeu** | 6 |
| 2.1 La borne supérieure des placements des bateaux | 6 |
| 2.2 Le nombre de placements dans la grille | 6 |
| 2.3 Le Calcul du nombre de façons de placer une liste de bateaux sur une grille | 8 |
| 2.4 Le nombre de répétitions avant d’obtenir deux grilles égales | 9 |
| 2.5 Approximation | 11 |
| **3. Modélisation probabiliste de jeu** | 15 |
| 3.1 Espérance théorique | 16 |
| 3.2 Bataille | 17 |
| 3.3 Joueur Aléatoire | 18 |
| 3.4 Joueur Heuristique | 19 |
| 3.5 Joueur probabiliste simpliﬁée | 20 |
| **4. Senseur imparfait** | 21 |
| 4.1 Quelle est la loi de probabilité de Yi et Zi|Yi ? | 22 |
| 4.2 Quelle est la probabilité d’une recherche infructueuse ? | 24 |
| 4.3 Mise à jour des probabilités πk | 25 |
| 4.4 Mise à jour des probabilités πi avec i ≠ k | 26 |
| 4.5 Algorithme de recherche | 27 |
| 4.6 Tests de l’algorithme de recherche d’objet | 28 |
| **Conclusion** | 29 |

# Introduction

L’objectif de ce projet est d’étudier le jeu bataille navale d’un point de vue probabiliste. Dans ce jeu 2 adversaires s’affrontent. Chaque joueur a une grille en forme d’une matrice 10x10 dans laquelle chaqu’un place ses bateaux. Chaque joueur peut placer 5 types de bateaux : un porte-avion qui occupe 5 cases, un croiseur qui occupe 4 cases, un contre-torpilleur qui occupe 3 cases, un sous-marin qui occupe 3 cases et un torpilleur qui occupe 2 cases. Les bateaux peuvent se placer horizontalement et verticalement dans la matrice des possibilités. Chaque jouer lance, à son tour, un tir. A chaque tir, le jouer choisi une case, puis son adversaire répond, soit par vide, touché ou coulé. C’est alors son tour de jouer. Le jeu s’arrête lorsque l’un des joueurs a coulé tout les bateaux de son adversaire. Il gagne alors la partie.

L’objectif du jeu consiste donc à trouver la meilleure stratégie qui va permet de gagner la partie avec le nombre minimum de tirs. Nous allons évaluer trois types de stratégies : une stratégie *aléatoire*, une stratégie *heuristique* (ie explorant les coups précédents et prenant une décision selon un certain nombre de règles) et une stratégie *probabiliste simplifiée*. Pour chaque stratégie, nous évaluerons leur efficacité, mesurée par le nombre de tirs nécessaire pour gagner la partie.

Nous avons structuré notre rapport de projet en 4 parties principales, correspondantes aux consignes du projet :

1. modélisation et fonctions simples,
2. combinatoire du jeu,
3. modélisation probabiliste de jeu et
4. senseur imparfait.

Il s'agir dans un premier temps d'étudier la combinatoire du jeu, puis ensuite de proposer une modélisation du jeu qui permet d'optimiser les chances de gagner et enfin d’évaluer une variante plus réaliste du jeu.

Ce projet a été mené à l’aide de Python. Le code que nous avons développé définit toutes les fonctions requises pour ce projet, notamment l’exécution du jeu, en plaçant les bateaux de manière aléatoire, vérifier les positions possibles, afficher les grilles (à l'aide de matplotlib.pyplot.imshow), l’évaluation probabiliste les diverses stratégies et l’analyse du cas plus général de la recherche d’un objet dans une région de l’espace prédéfinie.

# 1. Modélisation et fonctions simples

Dans ce chapitre nous présentons les fonctions de base développées dans notre code.

## 1.1 Le champ de bataille

Pour la création de la grille du jeu, nous avons crée un classe *Battlefield* qui gère le champ de bataille (une matrice d’entier de 10 \* 10 initialisé à 0) des joueurs.

## 1.2 La fonction placement des bateaux

Concernant le placement des bateaux dans la grille, nous avons définie 2 fonctions : la fonction *is\_empty()* qui prend en argument la grille, l’identifiant du bateau, la position qu’on souhaite placer et la direction. Elle renvoie une valeur booléenne. Le bateau peut être placé si et seulement si tous les bateaux sont vides et dans la limite de la grille. La fonction *place*() prend en argument les mêmes que *is\_empty*(). Elle teste si le bateau est plaçable dans la grille puis va mettre la tête du bateau en position et elle continue à placer dans le sens de la direction.

## 1.3 La fonction équivalence des grilles

La fonction *\_\_equal\_\_*() compare 2 grilles et renvoie *true* si elles ont le même contenu, ou *false* dans le cas contraire.

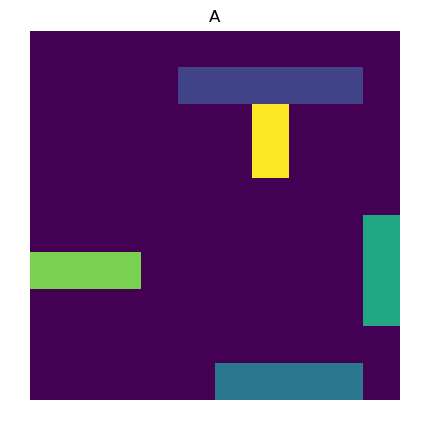
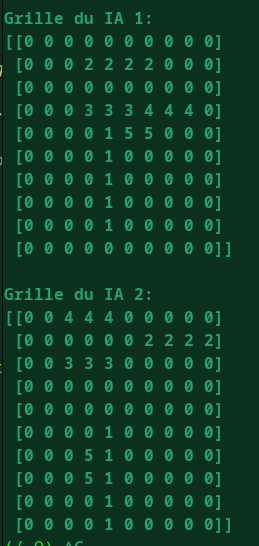
## 1.4 La génération de la grille

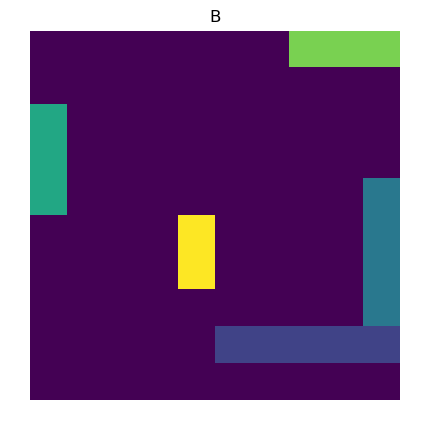
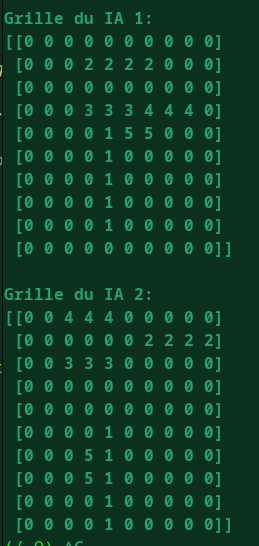
La fonction *random\_grid*() place les bateaux sur des positions aléatoires. Elle choisit aussi aléatoirement la direction (horizontale ou verticale).

## 1.5 L’affichage de la grille

La fonction *show*() affiche les grilles de jeu. La figure suivante montre le résultat de la génération d’une grille aléatoire pour chacun des 2 joueurs, graphiquement et sous forme matricielle.

**Figure 1**

****

****

# 2. Combinatoire du jeu

Dans ce chapitre nous répondons aux questions de l’énoncé concernant le dénombrement du nombre de grilles possibles dans différentes conditions, pour appréhender la combinatoire du jeu.

## 2.1 La borne supérieure des placements des bateaux

Afin de répondre à la première question de l’énoncé nous procédons d’abord de manière manuelle. Nous savons qu’il n’est pas possible de calculer de manière exacte le nombre de combinaisons possibles des bateaux dans la matrice. Toutefois, on peut, par approximation, calculer une borne supérieure.

Afin de calculer manuellement une borne supérieure nous partons des combinaisons possibles, type de bateau par type de bateau. Le tableau suivant présente les résultats des calculs :

|  |  |  |
| --- | --- | --- |
| **Taille du bateau** | **Calcul** | |
| 5 | (10 – 5) + 1 = 6 | 6 \* 10 \* 2 = 120 |
| 4 | (10 – 4) + 1 = 7 | 7 \* 10 \* 2 = 140 |
| 3 | (10 – 3) + 1 = 8 | 8 \* 10 \* 2 = 160 |
| 2 | (10 – 2) + 1 = 9 | 9 \* 10 \* 2 = 180 |
| 120\*140\*160\*160\*180 = 77.414.400.000 | | |

Nous obtenons 77.414.400.000 combinaisons possibles de 5 bateaux. Cela correspond donc aux résultats obtenus manuellement pour la borne supérieure.

## 2.2 Le nombre de placements dans la grille

Nous allons maintenant calculer, via notre code Python, le nombre de façons de placer un bateau donné sur une grille vide. D’abord nous devons tenir compte des dimensions de la grille, de la longueur du bateau, et des deux directions possibles car nous savons que le nombre de façons de placer un bateau sur la grille est déterminé par le nombre de positions valides pour le bateau dans la direction horizontale et le nombre de positions valides pour le bateau dans la direction verticale.

Nous avons procédé de la manière suivante :

* Pour une direction horizontale (direction 1), le bateau doit occuper une ligne donnée, mais il ne peut pas dépasser la grille en largeur. Donc, pour chaque ligne, il y a *GRID\_SIZE – ship\_length + 1* positions possibles sur les colonnes.
* Par exemple, pour un bateau de longueur 5 sur une grille de 10x10, on peut le placer dans 6 positions sur chaque ligne (0 à 5), car il ne peut pas dépasser la 6e colonne.
* Pour une direction verticale (direction 2), c’est similaire, sauf que le bateau ne peut pas dépasser la grille en hauteur. Pour chaque colonne, il y a *GRID\_SIZE – ship\_length + 1* positions possibles sur les lignes.

La somme des deux cas (horizontal et vertical) donne le nombre total de façons de placer le bateau sur la grille. Par exemple, pour un porte-avions (longueur 5) sur une grille de 10x10, l'exécution de ce code donne : Nombre de façons de placer le bateau 1 = 120. Cela signifie qu'il y a 120 façons de placer un porte-avions sur une grille vide, en tenant compte des deux directions (horizontale et verticale).

Pour calculer **le nombre total de façons de placer tous les bateaux** sur une grille vide, il faut calculer le nombre de façons de placer chaque bateau individuellement (comme nous l'avons fait précédemment pour un bateau) et ensuite multiplier toutes les possibilités entre elles. Cependant, il est important de noter que chaque bateau est placé indépendamment des autres. Autrement dit, chaque bateau peut être placé dans l'une des différentes orientations et positions, sans tenir compte des positions des autres bateaux (car la grille est vide).

Pour chaque bateau dans la liste des bateaux, nous utilisons la fonction *nombre\_de\_facons\_placer* pour obtenir le nombre de façons de placer ce bateau sur la grille. Ensuite, nous multiplions ces résultats pour obtenir le nombre total de combinaisons possibles de placement pour tous les bateaux.

**La fonction** *nombre\_de\_facons\_placer*, calcule le nombre de façons de placer un seul bateau (compte tenu des directions horizontales et verticales).

Le produit de toutes ces façons pour chaque bateau donne le nombre total de combinaisons de placements possibles. Si on demande les calcules pour chaque bateau, le nombre total de placements qui renvoi notre code est = 120\*140\*160\*160\*180 = 77 414 400 000. En conclusion, les résultats de la fonction codée viennent confirmer les résultats que nous avons obtenus manuellement.

## 2.3 Le Calcul du nombre de façons de placer une liste de bateaux sur une grille

Pour implémenter une fonction qui calcule le nombre de façons de placer une **liste de bateaux** sur une grille vide, nous allons généraliser la logique précédente pour prendre en compte plusieurs bateaux.

Cette fonction va parcourir chaque bateau de la liste, calculer le nombre de façons de placer ce bateau sur la grille vide en utilisant la fonction *nombre\_de\_facons\_placer* et enfin multiplier les résultats pour obtenir le nombre total de façons de placer tous les bateaux de la liste.

**Fonction** *nombre\_facons\_liste\_bateaux* :

* Cette fonction prend une grille vide et une liste de bateaux (leurs identifiants).
* Elle utilise la fonction *nombre\_de\_facons\_placer* pour calculer le nombre de façons de placer chaque bateau.
* Les résultats sont multipliés entre eux pour obtenir le nombre total de façons de placer tous les bateaux de la liste.

Nous testons la fonction avec des listes de bateaux contenant :

* 1 bateau (porte-avions),
* 2 bateaux (porte-avions et croiseur),
* 3 bateaux (porte-avions, croiseur, contre-torpilleur).

**Pour chaque liste** la fonction *nombre\_facons\_liste\_bateaux* calcule le produit des façons de placer chaque bateau de la liste. En exécutant ce code, voici les résultats attendus :

* *Nombre de façons de placer* [1]: 120
* *Nombre de façons de placer* [1, 2]: 120 \* 144 = 17280
* *Nombre de façons de placer* [1, 2, 3]: 120 \* 144 \* 168 = 2903040

Ce résultat montre qu'avec une grille vide et les bateaux placés indépendamment les uns des autres : Il y a 120 façons de placer le porte-avions, 17 280 façons de placer le porte-avions et le croiseur ensemble et environ 2 903 040 façons de placer le porte-avions, le croiseur et le contre-torpilleur ensemble. Ces valeurs représentent toutes les combinaisons possibles de placement pour chaque configuration de bateaux.

## 2.4 Le nombre de répétitions avant d’obtenir deux grilles égales

Pour créer une fonction qui génère des grilles aléatoires jusqu'à ce qu'une grille générée soit identique à une grille donnée, voici comment nous pouvons procéder :

1. **Génération aléatoire de grilles** : Nous utiliserons la fonction *genere\_grille* pour générer des grilles aléatoires, où les bateaux sont placés de manière aléatoire.
2. **Comparaison des grilles** : Nous utiliserons la fonction *eq* (que nous avons définie précédemment) pour comparer la grille générée avec la grille donnée en paramètre.
3. **Comptage des tentatives** : Nous compterons le nombre de grilles générées jusqu'à ce que l'une d'entre elles soit égale à la grille cible.
4. **Retour du nombre de tentatives** : Lorsque la grille générée correspond à la grille passée en paramètre, la fonction retourne le nombre de grilles générées.

### Dans notre code :

1. *eq*(grilleA, grilleB) : Cette fonction compare deux grilles pour vérifier si elles sont identiques. Elle utilise *numpy.array\_equal* pour comparer les matrices de manière élément par élément.
2. *genere\_grille*() : Cette fonction génère une grille aléatoire avec les bateaux placés de manière aléatoire. Elle utilise la fonction *place\_alea* pour placer chaque bateau sur la grille.
3. *cherche\_grille\_identique*(grille\_cible) :
   * Cette fonction génère des grilles aléatoirement en appelant *genere\_grille*().
   * Chaque grille générée est comparée à la grille cible passée en paramètre.
   * Si les deux grilles sont identiques (en utilisant *eq*), la fonction arrête la génération et retourne le nombre de grilles générées.
   * Le compteur garde une trace du nombre de tentatives nécessaires avant de trouver la grille cible.

### Ainsi, une grille cible est générée aléatoirement avec *genere\_grille*(). La fonction *cherche\_grille\_identique* génère des grilles jusqu'à ce qu'une grille identique à la grille cible soit trouvée. Le nombre de tentatives est affiché à la fin. A noter que le nombre de grilles générées avant de trouver une correspondance peut être très élevé, car chaque grille est générée de manière complètement aléatoire.

## 2.5 - Approximation

Pour approximer le **nombre total de grilles possibles** pour une liste de 5 bateaux, nous devons prendre en compte toutes les configurations possibles pour placer chaque bateau sur une grille de taille donnée (ici, 10x10). Puisque les bateaux ne doivent pas se chevaucher, les possibilités de placement d'un bateau dépendent des positions des bateaux précédents. L'idée est de commencer par calculer le nombre de placements possibles pour chaque bateau de manière approximative, en considérant les facteurs suivants :

* **Nombre de façons de placer chaque bateau individuellement** (comme calculé dans les exemples précédents).
* **Réduction progressive des cases disponibles à mesure que les bateaux sont placés**. Chaque bateau ajouté occupe une partie de la grille, réduisant ainsi l'espace disponible pour les bateaux suivants.

### Algorithme proposé :

1. **Calcul du nombre de façons de placer le premier bateau :**
   * Le premier bateau peut être placé sur une grille vide, donc nous pouvons utiliser le nombre de façons de le placer comme expliqué précédemment (sans restriction).
2. **Réduction de l'espace disponible** :
   * Pour chaque bateau suivant, nous devons réduire l'espace disponible. Cela se fait en fonction de la taille du bateau déjà placé, et de l'espace qu'il occupe sur la grille.
3. **Approximation** :
   * Chaque bateau réduit la grille proportionnellement à sa taille, donc à chaque étape, nous réduisons le nombre de positions disponibles en tenant compte de l'espace occupé par les bateaux déjà placés.
   * L'idée est d'ajuster les possibilités de placement en fonction de la probabilité de trouver des espaces libres pour les bateaux suivants.

### Dans notre code :

* *nombre\_de\_facons\_placer\_bateau* : Cette fonction calcule le nombre de façons de placer un bateau donné sur une grille partiellement remplie. Elle vérifie les espaces libres pour les placements horizontaux et verticaux.
* *approximer\_nombre\_total\_grilles* :
* Pour chaque bateau de la liste, elle calcule le nombre de façons possibles de le placer sur la grille.
* Après avoir calculé les façons de placer un bateau, elle le place aléatoirement pour simuler la réduction de l'espace disponible.
* Elle multiplie le nombre de façons de placer chaque bateau pour obtenir une approximation du nombre total de grilles.
* *place\_alea* : Cette fonction place un bateau de manière aléatoire sur la grille. Chaque bateau occupe certaines cases, réduisant ainsi l'espace disponible pour les suivants.

Nous générons une grille de manière aléatoire pour chaque bateau, en réduisant à chaque fois l'espace disponible. Cela nous permet de simuler progressivement le nombre total de façons possibles de placer tous les bateaux sur une grille sans se chevaucher.

Lorsque vous exécutez ce code avec la liste complète des 5 bateaux (porte-avions, croiseur, contre-torpilleur, sous-marin, et torpilleur), le programme retourne une estimation du nombre total de grilles possibles pour placer ces bateaux. Cette approche ne donne pas un résultat exact, mais elle fournit une estimation basée sur la réduction de l'espace disponible à chaque étape de placement des bateaux.

# 3 - Modélisation probabiliste de jeu

???

# 4 - Senseur imparfait

Ce chapitre vise à développer un algorithme qui permet de généraliser le problème à la recherche sur une région de l’espace plus vaste. L’espace de recherche est maintenant de N cellules et chaque cellule à une probabilité π i à priori de loger l’objet recherché. Yi est la variable aléatoire qui vaut 1 pour la cellule où se trouve l’objet et 0 partout ailleurs et Zi est le résultat de la recherche sur la case i, valant 1 si le résultat de la recherche est positif et 0 sinon. On suppose que chaque tentative de recherche est indépendante et identiquement distribué.

## 4.1 Quelle est la loi de probabilité de Yi et Zi|Yi ?

On définit *ps* comme étant la probabilité qu’une tentative de détecter l’objet dans une case soit fructueuse. *ps* se définit donc de la manière suivante :

*ps = P(Zi = 1 | Yi = 0)*

Considérons maintenant la probabilité simple de l’objet de se trouver dans la case i. Cette probabilité est exprimé par πi. Nous pouvons ainsi définit la loi de probabilité de Yi :

*P(Yi = 0) = 1 –* π*i*

*P(Yi = 1) =* π*i*

Yi étant la variable aléatoire qui associe à chaque case la valeur 1 si l’objet s’y trouve, 0 sinon.

Pour exprimer la loi de probabilité de Zi | Yi nous devons définir les 4 cas possibles :

1. La recherche ne détecte pas l’objet sur la case, et l’objet ne se trouvait pas dans la case. Dans ce cas la recherche est infructueuse. Nous écrivons alors :
   * *P(Zi = 0 |Yi = 0) = 1*
2. La tentative ne trouve pas l’objet sur la case et l’objet se trouve dans la case. Cela est exprimé par :
   * *P(Zi = 0 | Yi = 1) = 1 – ps*
3. La tentative détecte l’objet mais il n’est pas sur la case. Cela a un probabilité nule. Nous écrivons alors :
   * *P(Zi = 1 | Yi = 0) = 0*
4. La recherche détecte l’objet et il est effectivement sur la case. Cela correspond à la probabilité ps de détection sans erreur :
   * *P(Zi = 1 | Yi = 1) = ps*

## 4.2 Quelle est la probabilité d’une recherche infructueuse ?

Pour exprimer la probabilité que l’objet se trouve dans la case k d’une part, et que la recherche n’arrive pas à le détecter d’autre part, nous devons calculer : *P(Zk = 0) Ո (Yk = 1). Cela donne :*

*P(Zk = 0) Ո (Yk = 1) = P(Yk = 1) \* P(Zk = 0 | Yk = 1) =* π*k (1 - ps)*

## 4.3 Mise à jour des probabilités πk

Pour le calcul de la mise à jour des probabilité de π*k’ on doit calculer :*

π*k’ = P(Zk = 0 | Yk = 1) = P(Zk = 0) Ո (Yk = 1) / P(Zk = 0)*

Après quelques manipulations de cette expression on obtient le résultat suivant :

π*k’ = P(Zk = 0 | Yk = 1) = P(Zk = 0) Ո (Yk = 1) / P(Zk = 0) =*

*=* π *k (1 - ps) / (1-* π*k \* ps).*

## 4.4 Mise à jour des probabilités πi avec i ≠ k

Nous allons maintenant exprimer la formule de calcul des probabilités que l’objet se trouve se trouve dans les cases i telles que i ≠ k, c'est-à-dire le reste des cases où l’objet n’a pas encore été détecté. On cherche donc l’expression de π*i’, avec* i ≠ k :

π*i’ = P(Zi = 0 | Yk = 1) = P(Zk = 0) Ո (Yi = 1) / P(Zk = 0) )=*

*= P(Yi =1) \* P(Zk = 0 | Yi =1) / P(Zk = 0)*

Après quelques manipulations de cette expression on obtient le résultat suivant :

π*i’ = P(Yi = 1) \* P(Zk = 0 | Yi =1) / P(Zk = 0) =* π *i / (1 –* π*k \* ps)*

# 4.5 Algorithme de recherche

Les résultats précédents vont nous être très utiles pour définir notre algorithme de recherche. Notre algorithme de recherche peux s’exprimer sous la forme du pseudo-code suivant :

* *Génération de la grille de recherche*
* *Tant que l’objet n’est pas trouvé :*
* *Scanne la case ayant la plus grande probabilité* π*k de contenir l’objet*
  + - 1. *- Si la détection est fructueuse :*
    - *L’objet est trouvé*
* *Sinon* 
  + - *Mise à jour de la probabilité πk :*

*πk’ = πk(1 - Ps) / (1 - π k \* ps)*

* + - *Pour chaque case i différent de k :*
      * *Mise a jour de la probabilité πi :*

*πi’ = πi / (1 - π i \* ps)*

La probabilité *ps* que la recherche détecte avec succès l’objet et les probabilités π que chaque case contient l’objet sont fixées arbitrairement au départ.

Pour résoudre ce problème de recherche d'objet perdu dans un espace quadrillé en utilisant une approche bayésienne, nous pouvons suivre une méthode itérative basée sur le théorème de Bayes.

Nous avons implémenté un algorithme de recherche, basé sur le pseudo-code ci-avant. L'algorithme scanne la case ayant la plus grande probabilité d'abriter l'objet, puis met à jour les probabilités en fonction du succès ou de l'échec de la détection. La recherche continue jusqu'à ce que l'objet soit trouvé. Voici les étapes et l'algorithme proposés :

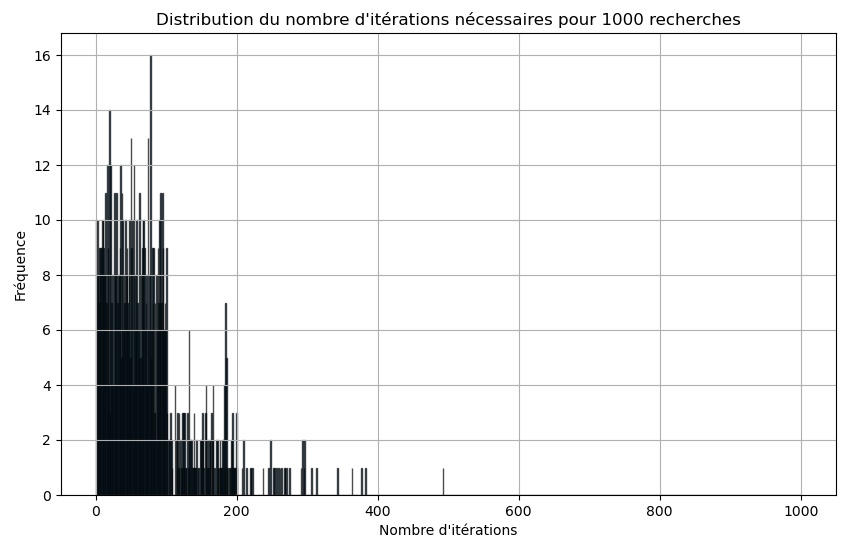
Nous avons commencé par générer une grille de taille N, avec une probabilité uniforme pour chaque case. L'objet est placé dans une case aléatoire à l'aide de la variable `*true\_position*`.

À chaque itération, on sélectionne la case ayant la plus grande probabilité. On simule la détection. Si la case scannée est la bonne et la détection réussit, l'algorithme s'arrête. A noter que l'algorithme s'arrête lorsque l'objet est détecté avec succès ou si le nombre maximal d'itérations est atteint.

Avec une grille de N = 100 (soit une grille 10x10) et une probabilité de détection ps = 0.8, l'algorithme effectue la recherche de l'objet. Si l'objet est trouvé avant le nombre maximal d'itérations, l'algorithme affiche la position et le nombre d'itérations nécessaires. Si l'objet n'est pas trouvé, l'algorithme renvoie un message d'échec.

Afin de comprendre et d'exécuter efficacement une recherche bayésienne dans un espace discret avec un capteur imparfait nous avons construit un histogramme qui permet de visualiser l'évolution des probabilités. La figure suivante présente l'histogramme qui représente la distribution du nombre d'itérations nécessaires pour chaque recherche lors de 1000 simulations. Chaque barre correspond au nombre d'itérations prises pour trouver l'objet dans une simulation particulière.

**Figure 2**

****

On observe que la majorité des recherches nécessitent un nombre modéré d'itérations, et la distribution est concentrée autour d'une valeur centrale, avec quelques recherches nécessitant plus d'itérations pour réussir. Cela montre l'efficacité de l'algorithme bayésien dans la plupart des cas, bien que certaines recherches puissent prendre plus de temps.

**4.6 - Tests de l’algorithme de recherche d’objet**

Pour étudier différentes distributions a priori, nous modifions l'initialisation des probabilités πi dans la fonction *bayesian\_search* pour considérer trois hypothèses différentes :

*Uniform\_prior* : toutes les cellules ont la même probabilité initiale. Cela correspond à notre cas précédent.

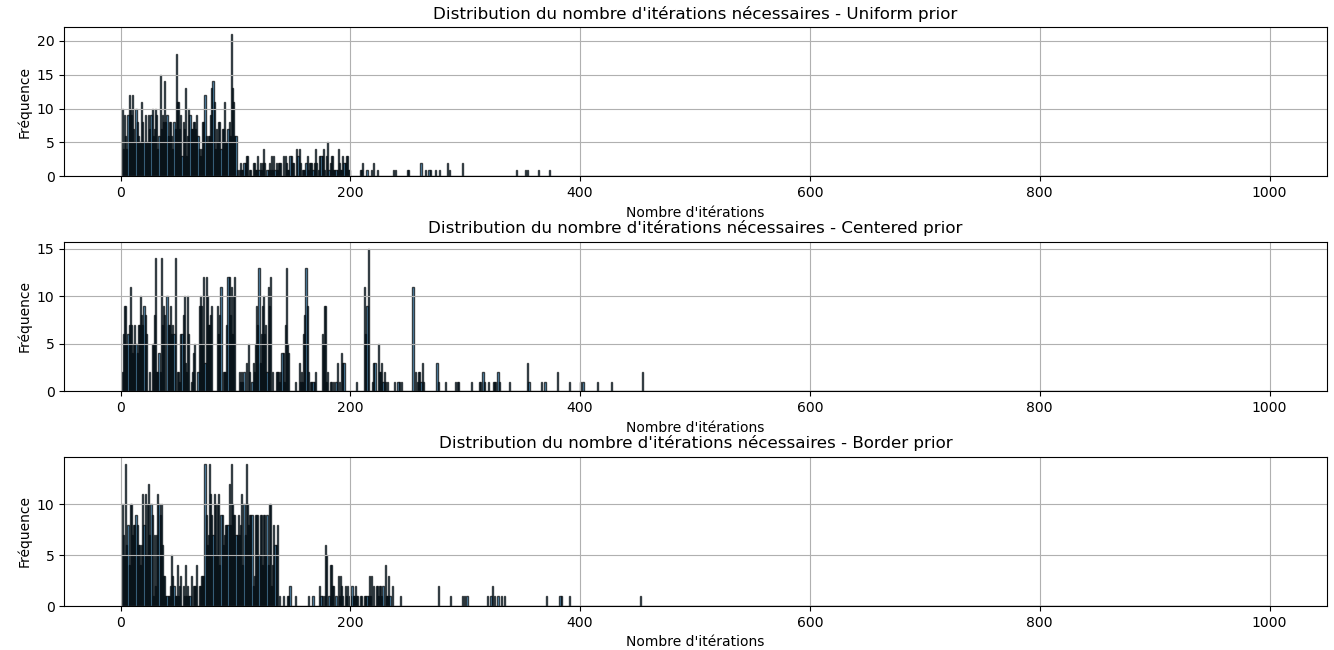
*Centered\_prior* : les cellules au centre de la grille ont une probabilité initiale plus élevée.

*Border\_prior* : les cellules sur les bords de la grille ont une probabilité initiale plus élevée.

Cela permet de tester la robustesse de l'algorithme face à différentes hypothèses sur la localisation probable de l'objet recherché et comparer les résultats obtenus dans chacune des hypothèses.

Afin de tester les 3 hypothèses différentes nous avons construit le schéma suivant.

**Figure 3**

****

Ce schéma présente les résultats obtenus sur 1000 recherches d’objets. Le premier graphique montre les résultats des simulations sur l’hypothèse à priori d’une distribution uniforme. Le deuxième graphique montre les résultats des simulations sur l’hypothèse à priori d’une distribution avec des probabilités plus élevés au centre de l’espace de cadrage. Le troisième graphique montre les résultats des simulations sur l’hypothèse à priori d’une distribution avec des probabilités plus élevés aux bords de l’espace de cadrage. La recherche est effectuée par les 3 algorithmes afin de comparer leur performance en relevant le nombre d’essais avant de détecter l’objet.

Sans surprise, chaque hypothèse affiche des résultats de simulations très différents. Même si dans le 3 cas les résultats montrent que l’algorithme est une bonne méthode de cadrage d’un espace de recherche, l’hypothèse de recherche non orientée (uniforme) est la moins performante. L’hypothèse de recherche avec des probabilités à priori plus élevées au centre semble être le plus performant dans la recherche de l’objet.

# Conclusion

L’objectif de départ de ce projet était d’abord d’étudier le jeu bataille navale d’un point de vue probabiliste pour ensuite élargir cette étude au cas plus général de recherche d’un objet sur un espace de sondage prédéterminé.

Da les chapitres 2 et 3 de l’analyse du probabiliste du jeu bataille navale nous avons proposé une modélisation qui nous a permit d’optimiser les chances d’un jouer d’trouver le plus vite, le premier, les bateaux positionnés par son adversaire.

Dans le chapitre 4 d’analyse d’un espace de sondage plus général, nous avons montré que l‘on peut améliorer les changes de trouver un objet plus rapidement si nous utilisons un algorithme qui s’appui sur des hypothèses de distribution des probabilités à priori qui sont plus élaboré qu’une distribution uniforme simple.