

---

## Carré Magique en Prolog

---

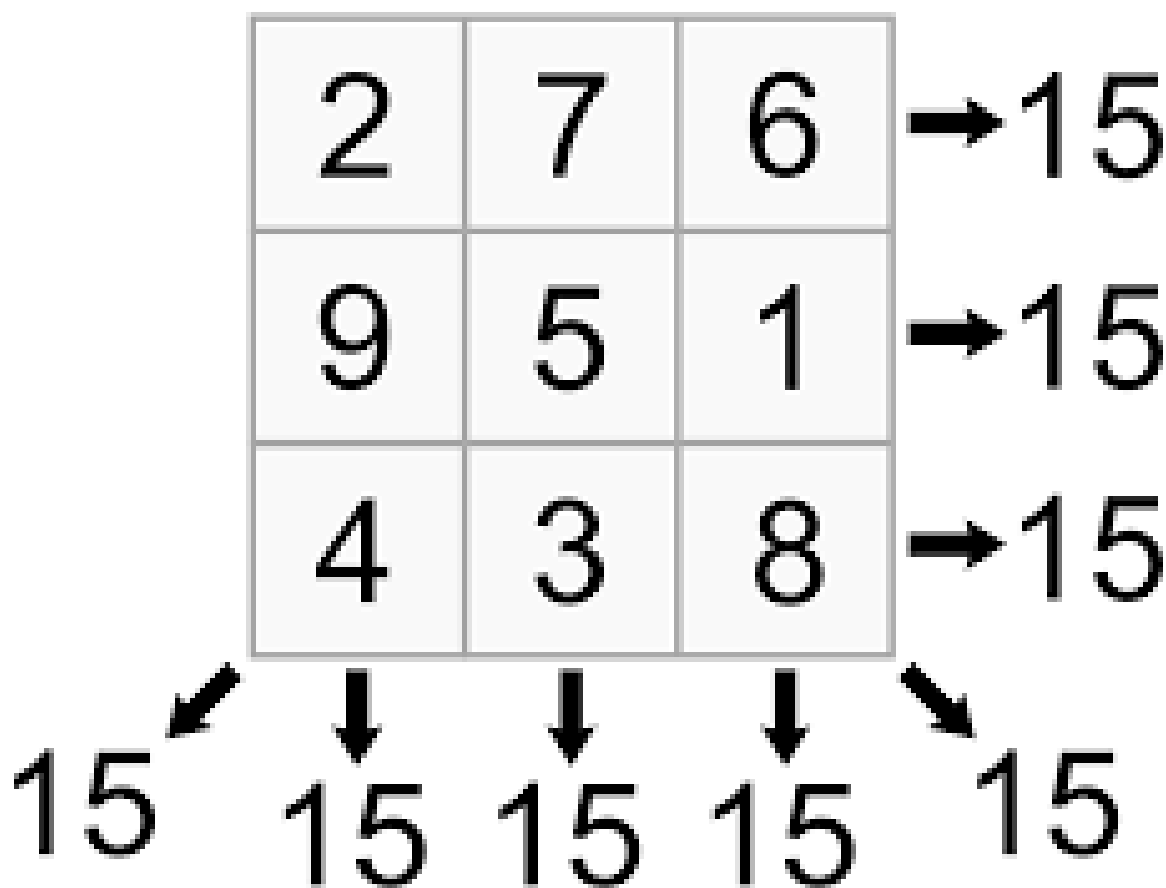


Figure 1: Carré magique

Réalisé par :  
Roxane LEDUC

Encadré par :  
Mme. CHAIGNAUD

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Conception et implémentation</b>	<b>3</b>
2.1	element_n/3 . . . . .	3
2.2	colonne_n/3 . . . . .	3
2.3	diag1 et diag2 . . . . .	4
2.4	toutes_les_listes/2 . . . . .	4
2.5	magique/1 . . . . .	5
2.6	genere_liste/2 . . . . .	5
2.7	retire_el/3 . . . . .	5
2.8	genere_ligne/4 . . . . .	5
2.9	genere_carre/3 . . . . .	6
2.10	mon_carre_magique/2 . . . . .	6
<b>3</b>	<b>Résultats</b>	<b>7</b>
<b>4</b>	<b>Conclusion</b>	<b>8</b>

# 1 Introduction

Dans le cadre d'un cours sur la Programmation Logique et par Contraintes, j'ai du réaliser un projet en Prolog. J'ai choisis d'implémenter des Carrés Magiques.

Les carrés magiques sont des figures géométriques fascinantes qui ont captivé l'attention des mathématiciens, des scientifiques et des amateurs de casse-tête pendant des siècles. Ils sont définis comme des arrangements de nombres entiers dans une grille carrée, où la somme de chaque ligne, colonne et diagonale principale est la même.

Dans ce projet, nous allons explorer les carrés magiques de tailles quelconques en utilisant le langage de programmation Prolog. Prolog est un langage de programmation logique qui se base sur la résolution de problèmes en trouvant des solutions à des requêtes logiques.

Nous allons explorer la conception et l'implémentation d'un programme Prolog pour générer et résoudre des carrés magiques de tailles quelconques. Nous allons utiliser des techniques de programmation logique pour générer des solutions uniques pour les carrés magiques, en garantissant que chaque carré magique est correct et valide.

Ce projet vise donc à explorer l'application des techniques de programmation logique pour résoudre des problèmes mathématiques complexes et à fournir une introduction pratique à la résolution de problèmes en Prolog. Il vise aussi à donner une appréciation plus profonde des carrés magiques, ainsi que des compétences en programmation logique utiles pour résoudre des problèmes mathématiques et logiques complexes.

## 2 Conception et implémentation

L'objectif final sera d'arriver à construire le prédicat `mon_carre_magique/2`. Pour ce faire, je vais détailler chacun des prédicats, un à un, qui ont permis son élaboration.

### 2.1 `element_n/3`

Le premier prédicat Prolog `element_n/3` est un prédicat récursif qui prend en entrée une liste et un entier `N` et qui unifie la variable `X` avec l'élément de la liste à la position `N`.

Le premier argument de `element_n/3` est l'entier `N`, qui est la position de l'élément que l'on souhaite extraire de la liste. Le deuxième argument est la liste elle-même, et le troisième argument est la variable `X` qui sera unifiée avec l'élément à la position `N`.

Le prédicat est défini en deux clauses. La première clause est une clause de base qui indique que si `N` est égal à 1, alors `X` est unifié avec le premier élément de la liste. Cette clause sert à définir le cas de base de la récursion, c'est-à-dire le cas où l'on cherche l'élément en position 1 dans la liste.

La deuxième clause est une clause récursive qui est utilisée lorsque `N` est supérieur à 1. Elle utilise la notation de liste Prolog `[T|Q]` pour décomposer la liste en deux parties : `T`, qui est la première élément de la liste, et `Q`, qui est le reste de la liste. La clause utilise ensuite la variable `M` pour calculer la position de l'élément que l'on cherche dans la nouvelle liste `Q`. La position `M` est égale à `N-1`, car on a déjà extrait le premier élément de la liste. On appelle ensuite le prédicat `element_n/3` récursivement avec les arguments `M`, `Q` et `X`, ce qui permet de chercher l'élément à la position `M` dans la liste `Q`. La variable `X` est ensuite unifiée avec l'élément trouvé, ce qui permet d'unifier la variable `X` avec l'élément à la position `N` dans la liste initiale.

Ainsi, ce prédicat permet d'extraire l'élément à une position donnée dans une liste en utilisant la récursion pour parcourir la liste.

```
element_n(1, [X|_], X).  
element_n(N, [_|Q], X) :- M is N-1, element_n(M, Q, X).
```

### 2.2 `colonne_n/3`

`colonne_n(N, Carre, C)` est défini en deux clauses. La première clause est une clause de base qui indique que si la matrice est vide, alors la colonne `N` est vide et `S` est unifiée avec une liste vide. Cette clause sert à définir le cas de base de la récursion, c'est-à-dire le cas où l'on a parcouru toutes les lignes de la matrice.

La deuxième clause est une clause récursive qui est utilisée lorsque la matrice n'est pas vide. Elle utilise, là-encore, la notation de liste Prolog `[L|R]` pour décomposer la matrice en deux parties : `L`, qui est la première ligne de la matrice, et `R`, qui est le reste de la matrice. La clause utilise ensuite le prédicat `element_n/3` pour extraire l'élément à la position `N` dans la ligne `L` et le stocker dans la variable `X`. Elle appelle ensuite récursivement le prédicat `colonne_n/3` avec les arguments `N`, `R` et `S`, ce qui permet de chercher la colonne `N` dans le reste de la matrice. La variable `X` est ensuite ajoutée à la liste `S`, qui contient les éléments de la colonne `N`.

Ainsi, ce prédicat permet d'extraire la colonne `N` d'une matrice en utilisant la récursion pour parcourir les lignes de la matrice et le prédicat `element_n/3` pour extraire les éléments de la colonne.

```

colonne_n(_, [], []).
colonne_n(N, [L|R], [X|S]) :- element_n(N, L, X), colonne_n(N, R, S).

```

## 2.3 diag1 et diag2

Le prédicat `diag1/2` est défini en deux clauses. La première clause appelle le prédicat `diag1/3` avec les arguments 1, Carre et D1. Cette clause sert à initialiser le premier argument I de `diag1/3` à 1.

La deuxième clause est le prédicat récursif `diag1/3` qui prend en entrée l'indice I, la matrice Carre, et la liste D1. La clause utilise la notation de liste `[L|R]` pour décomposer la matrice en une première ligne L et un reste de la matrice R. La clause utilise ensuite le prédicat `element_n/3` pour extraire l'élément X à la position I dans la ligne L. Cet élément est ajouté à la liste D1 grâce à la notation `[X|D1]`.

Le prédicat `diag1/3` est appelé récursivement avec l'indice Ipp qui est la valeur de I incremented de 1, et avec le reste de la matrice R. Cette récursion permet de parcourir toutes les lignes de la matrice et d'extraire tous les éléments de la première diagonale du carré.

La première clause sert à initialiser l'indice I à 1 pour commencer l'extraction de la première diagonale. La deuxième clause gère la récursion en extrayant l'élément de la diagonale et en appelant récursivement `diag1/3` pour extraire les éléments suivants de la diagonale.

Ainsi, ce prédicat utilise la récursion pour extraire les éléments de la première diagonale d'un carré.

On va fonctionner de la même manière pour déterminer la seconde diagonale, mais, on partira cette fois-ci du "bout" du carré (à l'aide de `dim/2`) et on décrémentera la valeur de I de 1.

```

diag1(Carre, D1) :- diag1(1, Carre, D1).
diag1(_, [], []).
diag1(I, [L|R], [X|D1]) :- element_n(I, L, X), Ipp is I+1, diag1(Ipp, R, D1).

dim([], 0).
dim([_|_], Dim) :- length(T, Dim).

diag2(Carre, D2) :- dim(Carre, Dim), diag2(Dim, Carre, D2).
diag2(_, [], []).
diag2(I, [L|R], [X|D2]) :- element_n(I, L, X), Imm is I-1, diag2(Imm, R, D2).

```

## 2.4 toutes\_les\_listes/2

Le prédicat `toutes_les_listes/2` combine les éléments de la matrice Carre, les colonnes de la matrice Carre, les éléments de la première diagonale, les éléments de la deuxième diagonale et les lignes de la matrice transposée T dans une seule liste X. Cette liste X contient toutes les listes possibles d'éléments du carré magique Carre. On fait donc ici appel à une fonction déjà codé en Prolog (`transpose/2`), d'où l'utilisation de la librairie `clpfd`. Le prédicat `transpose/2` permet de transposer la matrice Carre. Cela permet de récupérer toutes les colonnes du carré magique sous forme de lignes dans une nouvelle matrice T.

```

toutes_les_listes(Carre, X) :- transpose(Carre, T), diag1(Carre, D1),
    diag2(Carre, D2), append(Carre, [D1, D2], Y), append(Y, T, X).

```

```
?- toutes_les_listes([[1,2,3], [4,5,6], [7,8,9]], X).
X = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 5, 9], [3, 5, 7], [1, 4, 7], [2, 5|...], [3|...]]
```

Figure 2: Exemple: *toutes\_les\_listes/2*

## 2.5 magique/1

`magique(Carre)` renvoie `True` si le carre `Carre` est magique (il faut que toutes les composantes de `Comp` aient la même somme). Il utilise le prédicat `meme_somme/2` qui va calculer la somme `S` de la première sous-liste de `Comp` et la comparer à la somme de toutes les autres sous-listes de `Comp` en appelant récursivement `meme_somme/2`.

```
magique(Carre) :- toutes_les_listes(Carre, Comp), meme_somme(Comp, _).

meme_somme([], _).
meme_somme([L|R], S) :- sum_list(L,S), meme_somme(R, S).
```

## 2.6 genere\_liste/2

`genere_liste(N, L)` génère la liste des nombres allant de 1 à `N` dans la variable `L`. On peut noter que, dans la première clause, le `!"` (cut) permet d'arrêter la recherche de solutions alternatives et de s'assurer que cette clause est la seule qui sera utilisée pour la valeur `N=1`.

```
genere_liste(1, [1]) :- !.
genere_liste(N, [N|L]) :- Nmm is N-1, genere_liste(Nmm, L).
```

## 2.7 retire\_el/3

`retire_el(L, X, R)` retire la première occurrence de l'élément `X` trouvée dans la liste `L` et renvoie le résultat dans la variable `R`. Si l'élément `X` n'appartient pas à `L`, alors `R=L`.

```
retire_el([], _, []).
retire_el([X|Q], X, Q) :- !.
retire_el([T|R], X, [T|R]) :- retire_el(Q, X, R).
```

## 2.8 genere\_ligne/4

`genere_ligne(N, ListeNbs, L, NewListeNbs)` est capable de générer une liste de `N` nombres à partir de la liste `ListeNbs` en renvoyant le résultat dans la variable `L` et la liste des nombres non utilisés dans `NewListeNbs`.

La seconde clause spécifie que lorsque `N` est différent de 0, on doit choisir un élément `X` de la liste `"ListeNbs"`, le supprimer de cette liste et l'ajouter à la ligne en cours de construction `[X|R]`. Le prédicat `"member(X, ListeNbs)"` vérifie si l'élément `X` est bien dans la liste `"ListeNbs"`. Si c'est le cas, la fonction `"retire_el(ListeNbs, X, ListeNbs2)"` retire l'élément `X` de la liste `"ListeNbs"` et renvoie la liste résultante `"ListeNbs2"`. La variable `M` contient la valeur `N-1`, et la ligne suivante `"genere_ligne(M, ListeNbs2, R, NewListeNbs)"` appelle récursivement le prédicat `"genere_ligne"` avec la valeur `N-1` et la liste `"ListeNbs2"` mise à jour. La variable `R` contient la ligne de `N-1` nombres générée récursivement, et la variable `"NewListeNbs"` contient la liste `"ListeNbs2"` mise à jour.

```

genere_ligne(0, ListeNbs, [], ListeNbs) :- !.
genere_ligne(N, ListeNbs, [X|R], NewListeNbs) :- member(X, ListeNbs),
    retire_el(ListeNbs, X, ListeNbs2), M is N-1, genere_ligne(M,
    ListeNbs2, R, NewListeNbs).

```

```

?- genere_ligne(3, [1,2,5,7,8], L,N).
L = [1, 2, 5],
N = [7, 8]

```

Figure 3: Exemple: *genere\_ligne/4*

## 2.9 genere\_carre/3

Le prédicat "genere\_carre" génère un carré (pas forcément magique) de taille N en créant une liste de  $N^2$  nombres distincts, en utilisant la fonction "genere\_ligne" pour générer chaque ligne du carré, et en appelant récursivement "genere\_carre" pour générer chaque ligne suivante jusqu'à ce que le carré soit complet.

```

genere_carre([], _, []).
genere_carre(N, Carre) :- NN is N*N, genere_liste(NN, ListeNbs),
    genere_carre(ListeNbs, N, Carre).
genere_carre(ListeNbs, N, [L|S]) :- genere_ligne(N, ListeNbs, L,
    NewListeNbs), genere_carre(NewListeNbs, N, S).

```

## 2.10 mon\_carre\_magique/2

Ce dernier prédicat permet ainsi de générer tous les carrés magiques d'ordre N (on teste toutes les combinaisons et on s'assure que le carré soit magique).

```

mon_carre_magique(N,C) :- genere_carre(N,C), magique(C).

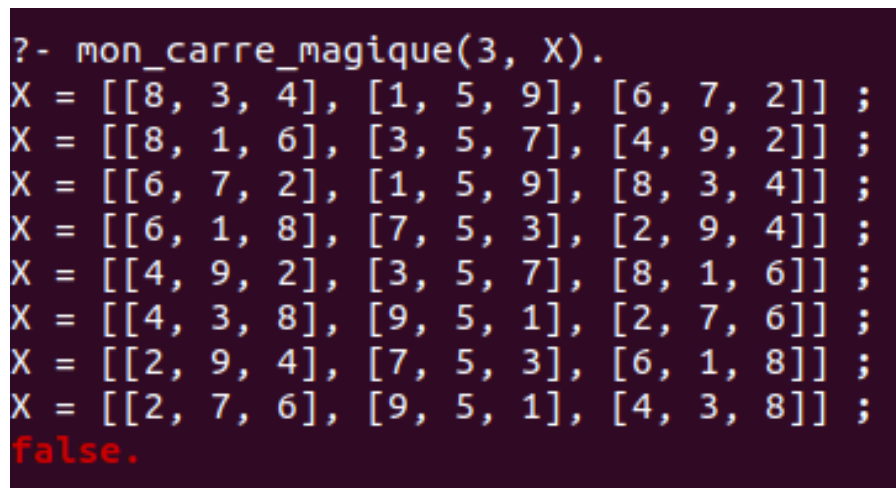
```

### 3 Résultats

Avant d'exécuter `mon_carre_magique(N, X)`, il ne faut pas oublier d'importer la librairie `clpfd` qui permet d'obtenir la transposé d'une matrice.

```
?- [carre_magique_1].  
?- use_module(library(clpfd)).  
?- mon_carre_magique(3, X).
```

Après avoir fait tourner le code, on obtient par exemple pour un carré de taille 3\*3, le résultat suivant:



```
?- mon_carre_magique(3, X).  
X = [[8, 3, 4], [1, 5, 9], [6, 7, 2]] ;  
X = [[8, 1, 6], [3, 5, 7], [4, 9, 2]] ;  
X = [[6, 7, 2], [1, 5, 9], [8, 3, 4]] ;  
X = [[6, 1, 8], [7, 5, 3], [2, 9, 4]] ;  
X = [[4, 9, 2], [3, 5, 7], [8, 1, 6]] ;  
X = [[4, 3, 8], [9, 5, 1], [2, 7, 6]] ;  
X = [[2, 9, 4], [7, 5, 3], [6, 1, 8]] ;  
X = [[2, 7, 6], [9, 5, 1], [4, 3, 8]] ;  
false.
```

Figure 4: Résultats pour un carré magique de taille 3\*3

On peut noter qu'il existe des dispositions magiques pour tout carré d'ordre  $n \geq 1$ . Le carré d'ordre 1 est trivial. Le carré d'ordre 2 est également trivial puisqu'il n'est possible qu'en répétant le même nombre dans les quatre cases. Le plus petit cas non trivial est le carré d'ordre 3 (à l'exception des carrés contenant partout le même nombre).

Il existe un seul carré magique parfait d'ordre 3, qui peut être obtenu en permutant les nombres dans un carré magique de base. Tous les carrés magiques d'ordre 3 qui ne sont pas des carrés magiques parfaits peuvent être obtenus en ajoutant ou en soustrayant un multiple de 3 à chaque nombre du carré magique parfait. Il y a donc huit carrés magiques d'ordre 3 qui ne sont pas des carrés magiques parfaits.

Par ailleurs, il existe 880 carrés magiques d'ordre 4 distincts, mais il n'y a que deux carrés magiques parfaits d'ordre 4 connus.

J'ai notamment appris que de manière générale, la constante magique d'un carré magique normal dépend uniquement de  $n$  et vaut :  $\frac{n(n^2+1)}{2}$ .



## 4 Conclusion

En conclusion, ce projet sur les carrés magiques de tailles quelconques implémentés en Prolog m'a permis d'explorer en profondeur les propriétés et les caractéristiques de ces figures géométriques fascinantes, ainsi que les techniques algorithmiques utilisées pour les résoudre.

J'ai également pu mettre en pratique mes compétences en programmation logique en implémentant un programme Prolog pour générer et résoudre ces carrés magiques, en utilisant des techniques de programmation logique pour garantir la validité et l'unicité de chaque solution.

En travaillant sur ce projet, j'ai développé ma capacité à résoudre des problèmes mathématiques et logiques complexes, ainsi qu'à concevoir et à implémenter des programmes informatiques pour résoudre ces mêmes problèmes.