OPTIMISATION DE TRAJECTOIRES DANS UN CHAMP DE VECTEURS

FARIEZ Axelle

SOMMAIRE

- Introduction
- Position du problème
- Modélisation d'une carte de vents
- Algorithme A*
- Modélisation : Trafic aérien
- Conclusion

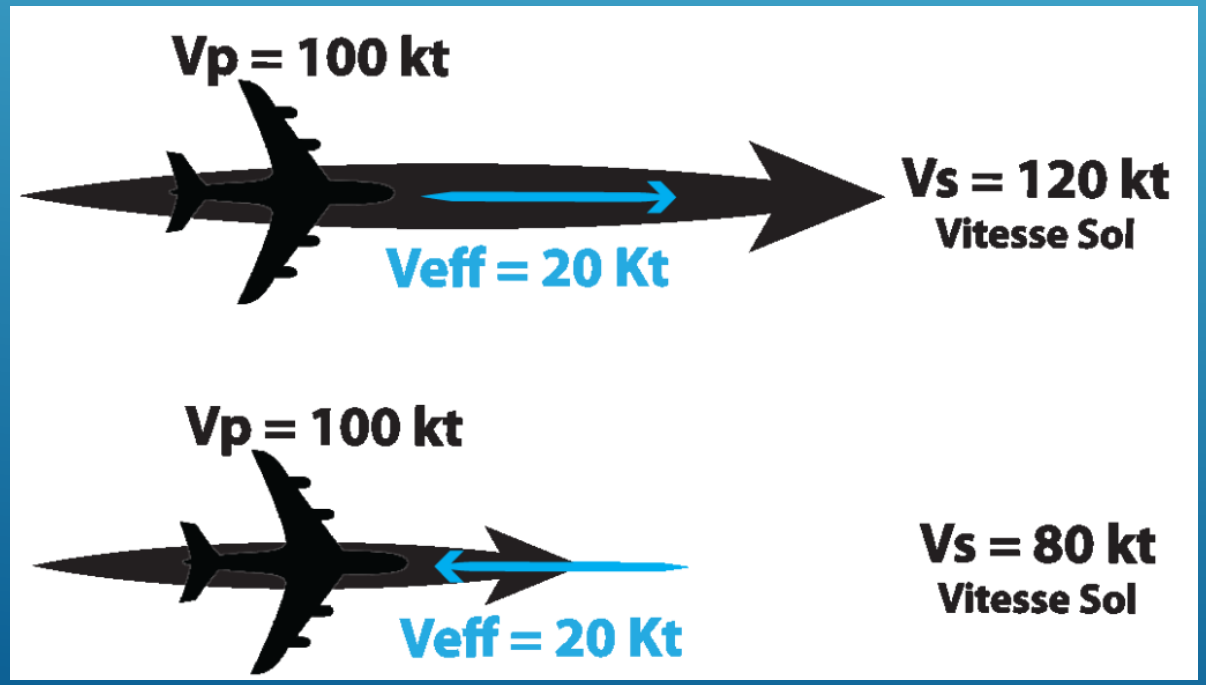
INTRODUCTION

- Le secteur aéronautique

- De nombreuses contraintes:
  - Sécurité
  - Vitesse
  - Coût
  - Environnement

POSITION DU PROBLÈME

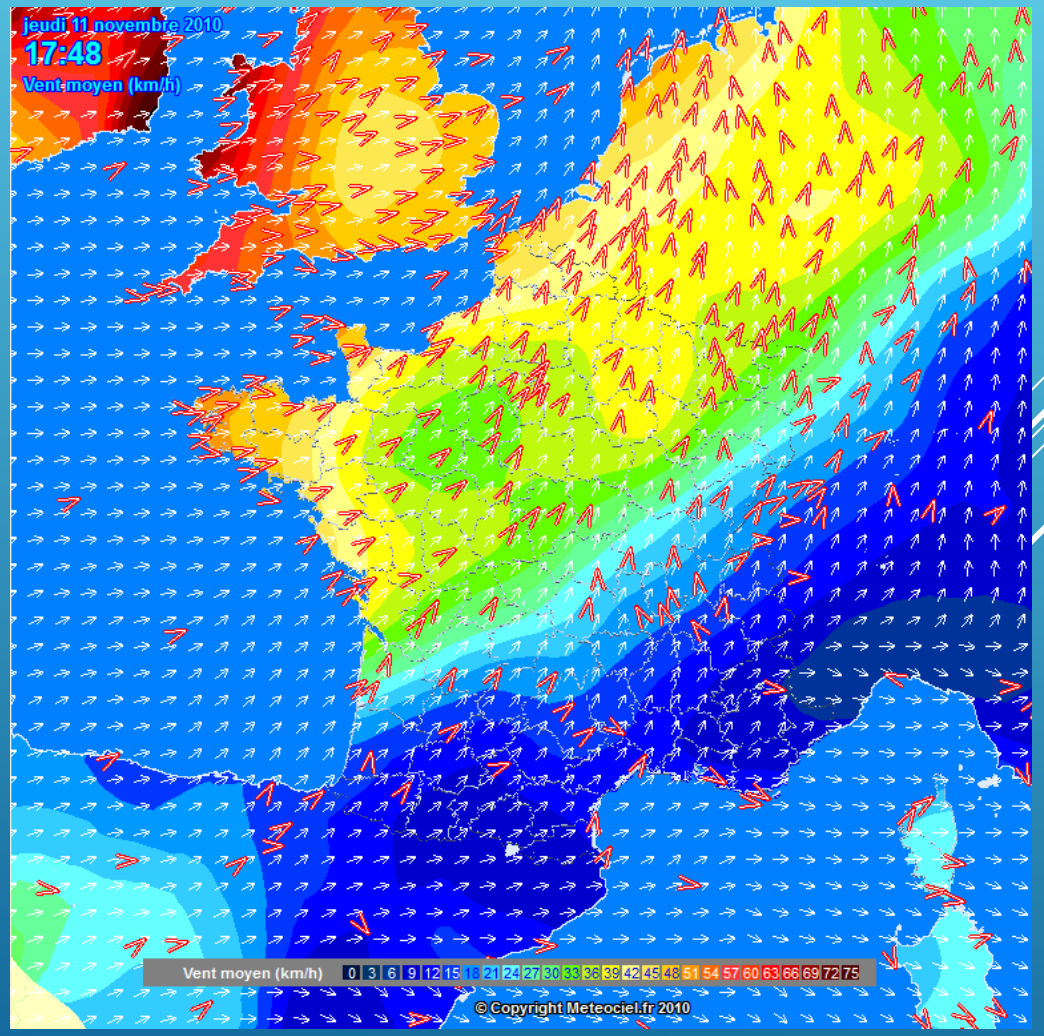
- L'influence des vents sur un trajet aérien



- Application réelle au trafic difficile

MODÉLISATION D'UNE CARTE DE VENTS

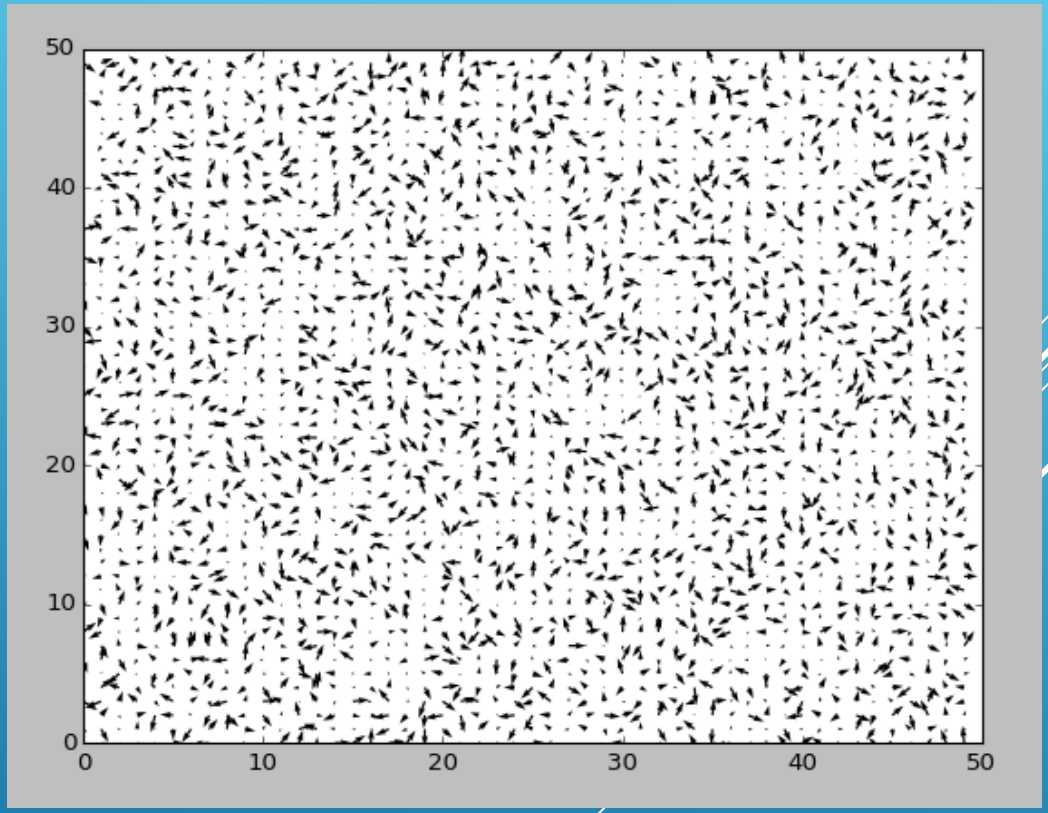
- Modèle : assimilation du vent à des vecteurs



MODÉLISATION D'UNE CARTE DE VENTS

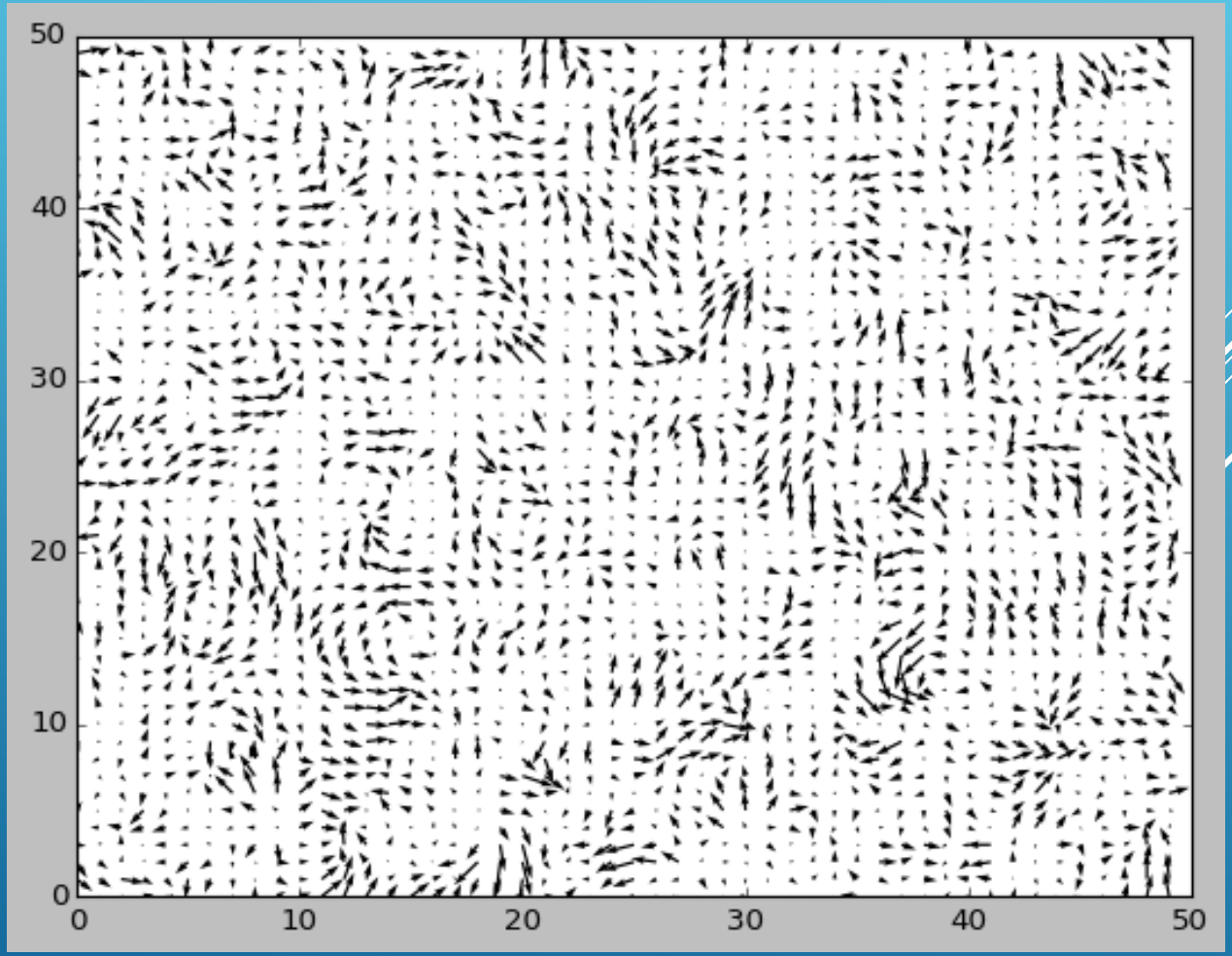
- Création d'un champ de vecteurs aléatoires par programmation informatique

```
angle=np.array([random() for i in range(N*N)])
intensite=np.array([10+90*random() for i in range(N*N)])
champ_y=intensite*np.sin(2*np.pi*angle)
champ_x=champ_x.reshape((N,N))
champ_y=champ_y.reshape((N,N))
```



MODÉLISATION D'UNE CARTE DE VENTS

- Lissage du premier champ de vecteur pour en obtenir un second, cohérent



MODÉLISATION D'UNE CARTE DE VENTS

- Méthode : calcul de la moyenne en chaque point

```
def voisins(matrice,i,j):    #relève les valeurs de tous les voisins d'une case donnée
    res=[matrice[i,j]]        #on crée une liste avec la valeur de la case (i,j)
    if i>0:                   #ajout des valeurs voisines de [i,j] de la ligne i-1
        res.append(matrice[i-1,j])
        if j>0:
            res.append(matrice[i-1,j-1])
        if j<N-1:
            res.append(matrice[i-1,j+1])
    if i<N-1:                      #ajout des valeurs voisines de [i,j] de la ligne i+1
        res.append(matrice[i+1,j])
        if j>0:
            res.append(matrice[i+1,j-1])
        if j<N-1:
            res.append(matrice[i+1,j+1])
    if j>0:                              #ajout des valeurs voisines de [i,j] de la ligne i
        res.append(matrice[i,j-1])
    if j<N-1:
        res.append(matrice[i,j+1])
    return res


for i in range(N):      #nouvelle matrice avec la moyenne des voisins (lissage)
    for j in range(N):
        inter_x[i,j]=moyenne(voisins(champ_x,i,j))
        inter_y[i,j]=moyenne(voisins(champ_y,i,j))
```

PATHFINDING : DIFFÉRENTS ALGORITHMES

- Algorithmes génétiques
- Algorithme de Dijkstra
- Algorithmes Métaheuristiques (algorithme A*)

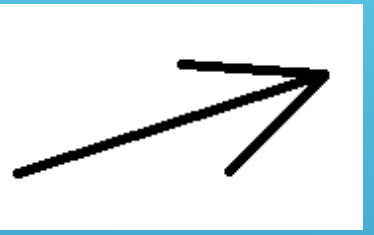
ALGORITHME A*

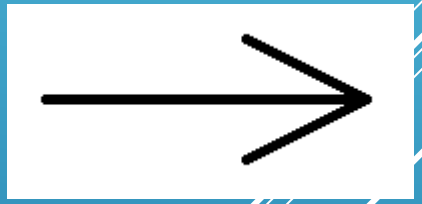
- Heuristique : distance à vol d'oiseau
- Calcul du coût: Produit scalaire vent-avion multiplié par la distance à parcourir

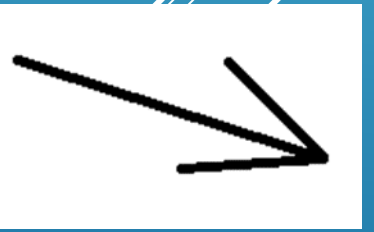
```
def heuristic(start_node,final_node):
    x,y=start_node
    xf,yf=final_node
    H=((x-xf)**2+(y-yf)**2)**(1/2)
    return H
```

ALGORITHME A*

- Définition des nœuds voisins







ALGORITHME A*

- Création d'une liste ouverte, et d'une liste fermée
- Choix du nœud actuel
- Définition de trois listes de valeurs: cameFrom, gScore, fScore

```
def trouve_indice_min(lst, fScore):
    z = lst[0]
    mini = fScore[z[0]][z[1]]
    i_mini = 0
    for i, z in enumerate(lst):
        nouveau_mini = fScore[z[0]][z[1]]
        if nouveau_mini < mini:
            mini = nouveau_mini
            i_mini = i
    return i_mini
```

ALGORITHME A*

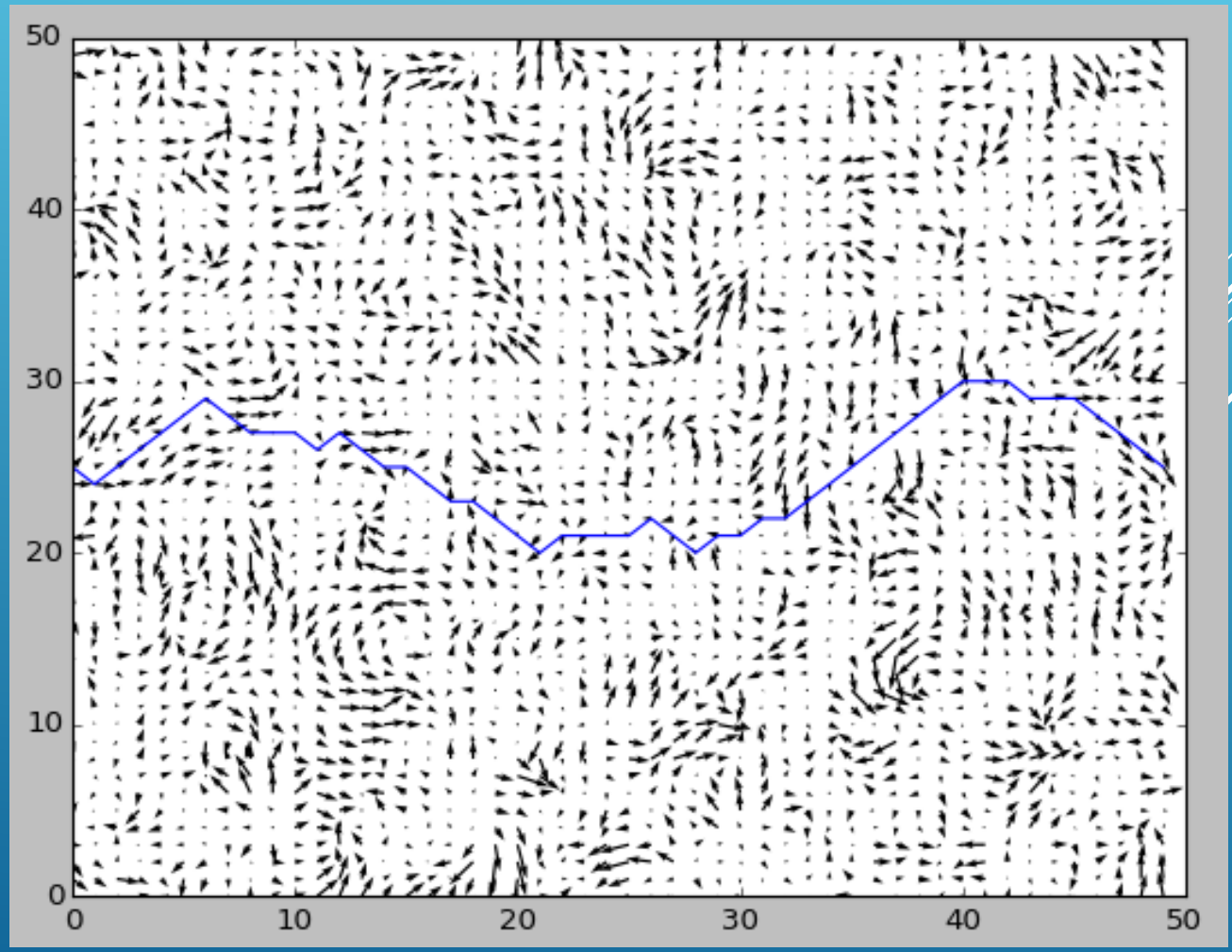
- Etude des nœuds voisins du nœud actuel

```
del openlist[i]
closedlist.append(current)
for neighbor in noeuds_voisins(current):
    x, y = neighbor
    if not neighbor in closedlist:
        # The distance from start to a neighbor
        tentative_gScore = gScore[current[0]][current[1]] + cout(x, y)
        A = node_in_L(neighbor,openlist)
        if not A:      # Discover a new node
            openlist.append(neighbor)
        if A or tentative_gScore < gScore[x][y]:
            cameFrom[x][y] = current
            gScore[x][y] = tentative_gScore
            fScore[x][y] = gScore[x][y] + heuristique(x, y)
```

ALGORITHME A*

- Construction du chemin final

```
def reconstruct_path(cameFrom,current):
    total_path = [current]
    while current != (xd,yd):
        current = cameFrom[current]
        total_path.append(current)
    return total_path.reverse
```

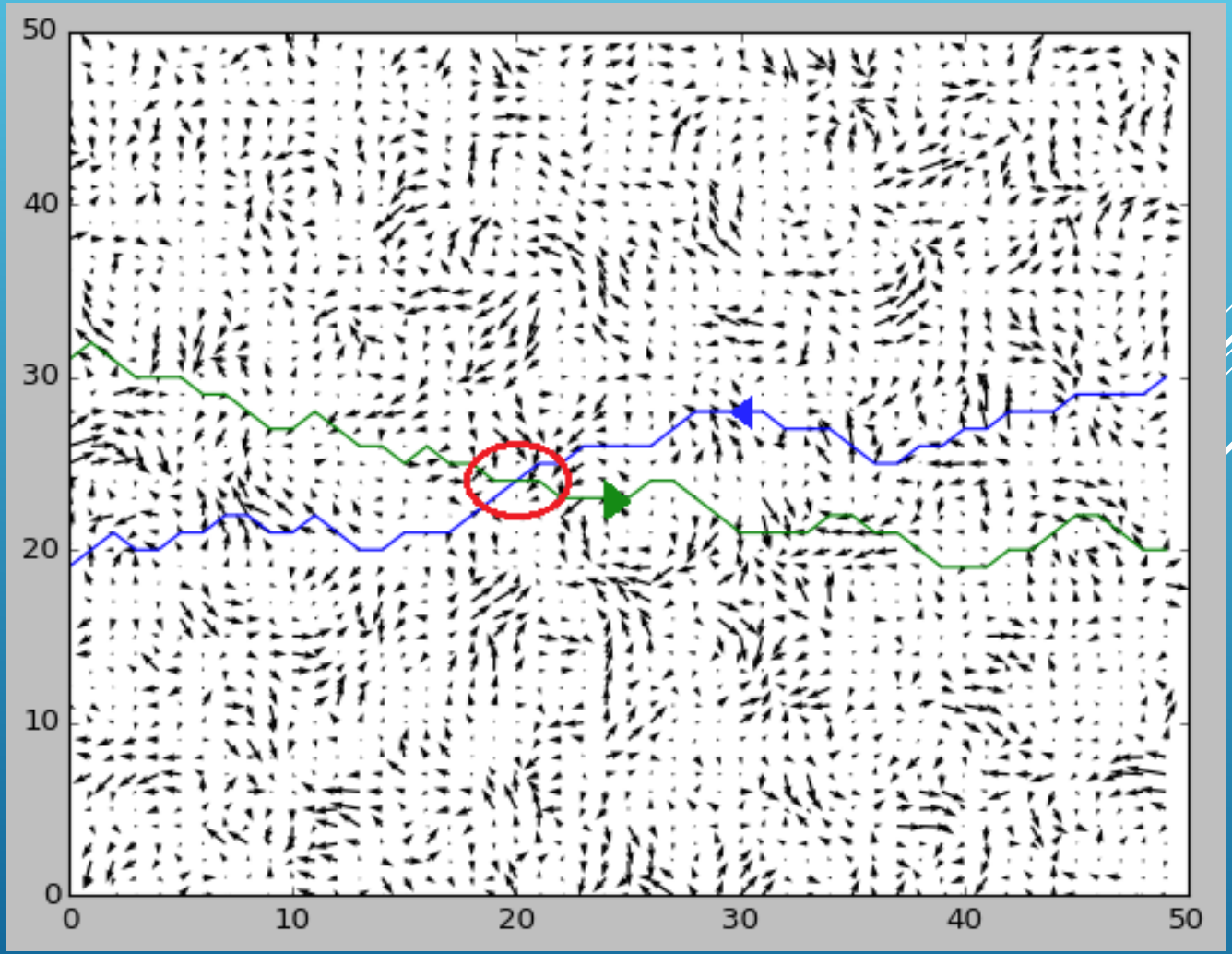


TRAFIC AÉRIEN

- Plans de vol
- Détermination des trajectoires :
  - Routes fixes
  - Zones de conflits à éviter
- Modifications exceptionnelles des trajectoires

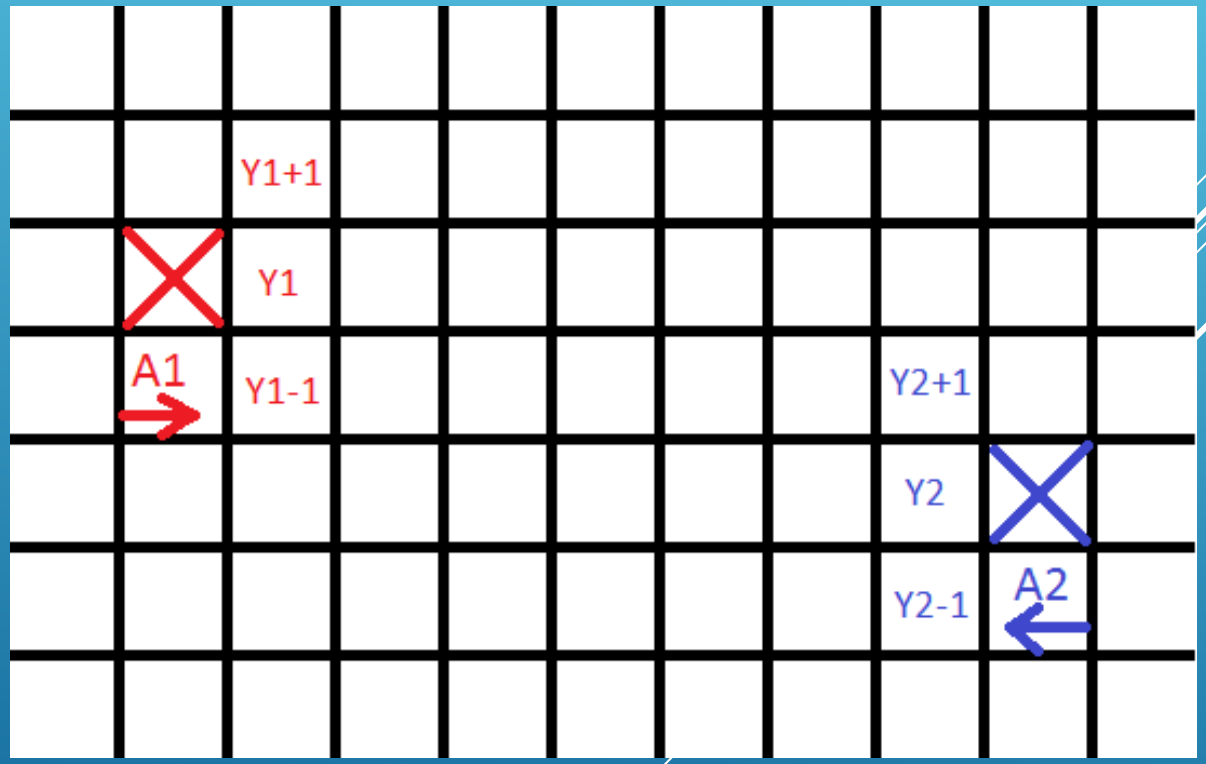
TRAFIC AÉRIEN

- Application de l'algorithme sur deux avions
- Problème : risque de collision



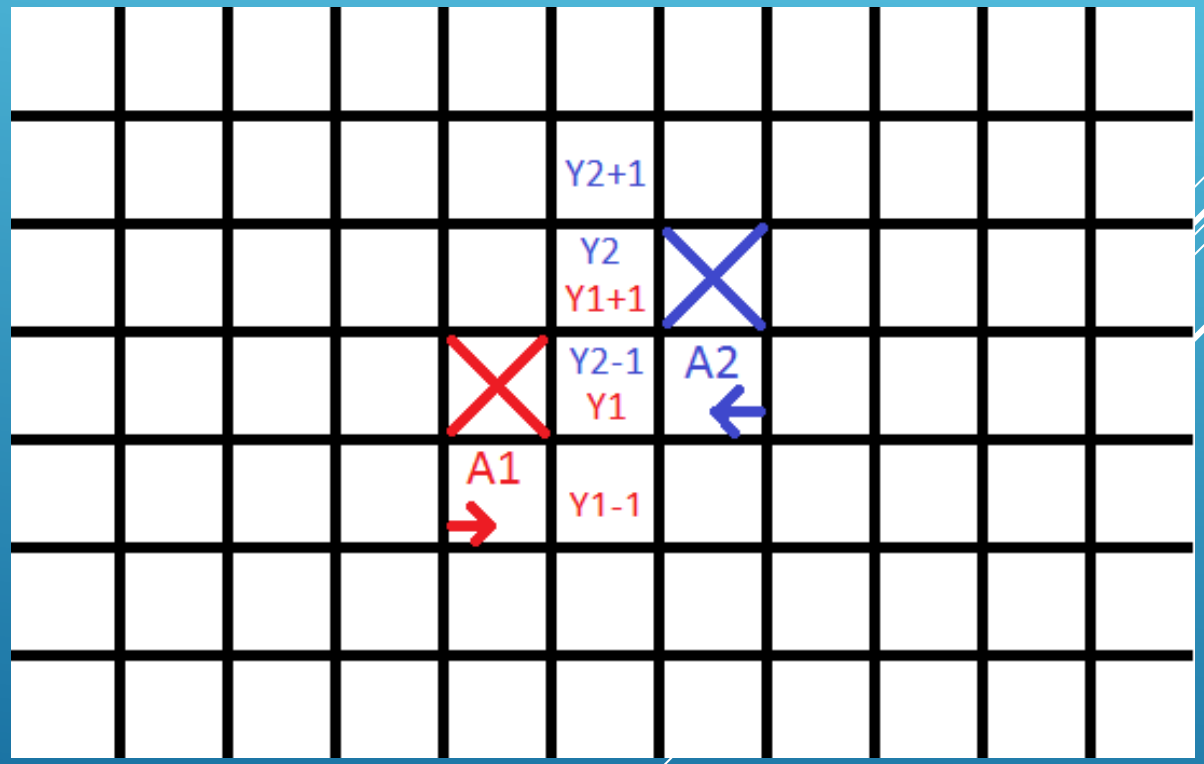
TRAFIC AÉRIEN

- Mise en situation : deux avions identique se déplaçant en sens inverse en utilisant les vents
- Etude de différents cas



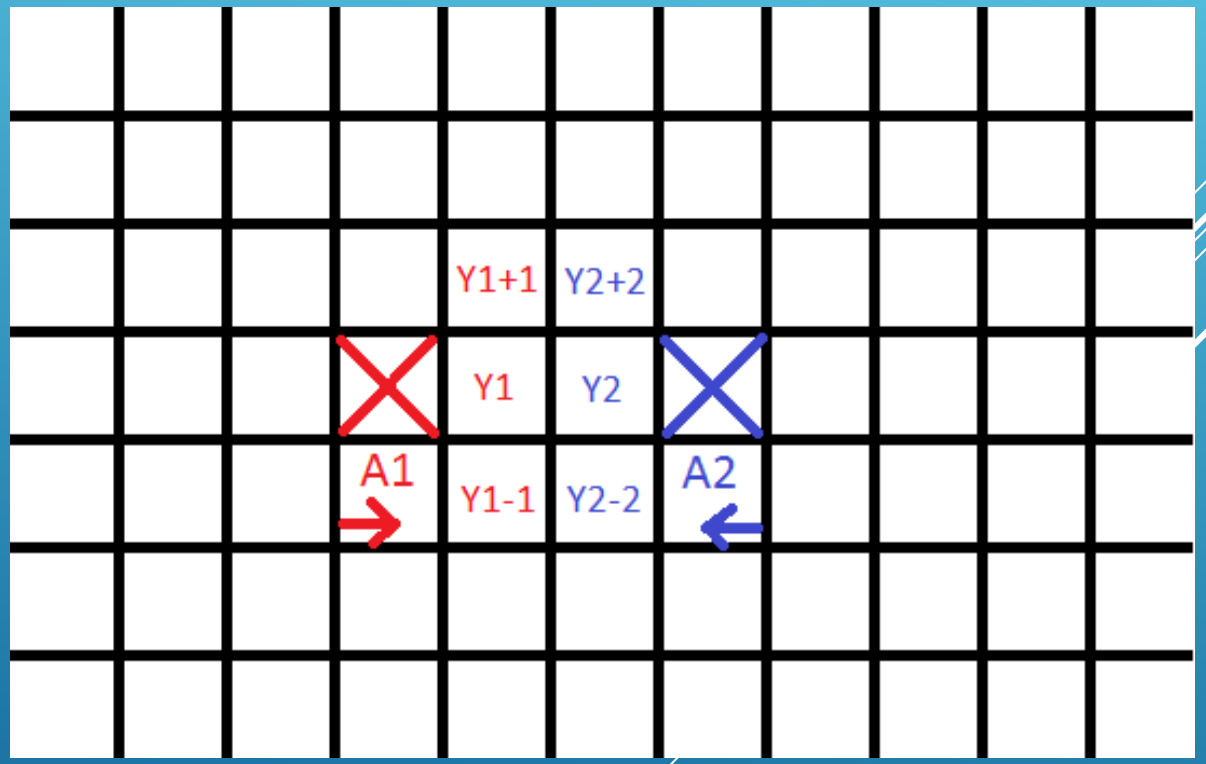
TRAFIC AÉRIEN

- Cases de trajectoires communes
- Risques de collisions directes



TRAFIC AÉRIEN

- Risques de collisions au prochain déplacement
- Nécessité des distances de sécurité



CONCLUSION

- Le projet:
  - Optimisation de la vitesse du trajet d'un avion
  - Etude des problèmes liés au trafic aérien pour améliorer la sécurité
- En réalité:
  - Frottements de l'air
  - Vents évoluant au cours du déplacement
  - Niveaux de vol

ANNEXE 1 : NŒUDS VOISINS

```
def neighbors(current_node):    #Séléctionne les voisins possibles du noeud actuel
    x,y=current_node
    if y==49:
        L=[(x+1,y),(x+1,y-1)]
    elif y==0:
        L=[(x+1,y+1),(x+1,y)]
    else:
        L=[(x+1,y+1),(x+1,y),(x+1,y-1)]
    return L
```

ANNEXE 2 : COÛT

- Produit scalaire entre la vitesse d'arrivée de l'avion et celle du vent
- Multiplié par la distance à parcourir

```
def cout(x,y):
    A=speed_A380
    V=norme(current)
    W=A*V*np.cos(angle(current))
    if current==(x-1,y):
        teta=0
    elif current==(x-1,y-1):
        teta=-1
        W=W*cos(teta)*(2**(1/2))
    elif current==(x-1,y+1):
        teta=1
        W=W*cos(teta)*(2**(1/2))
    return W
```

ANNEXE 3 : PLUSIEURS AVIONS