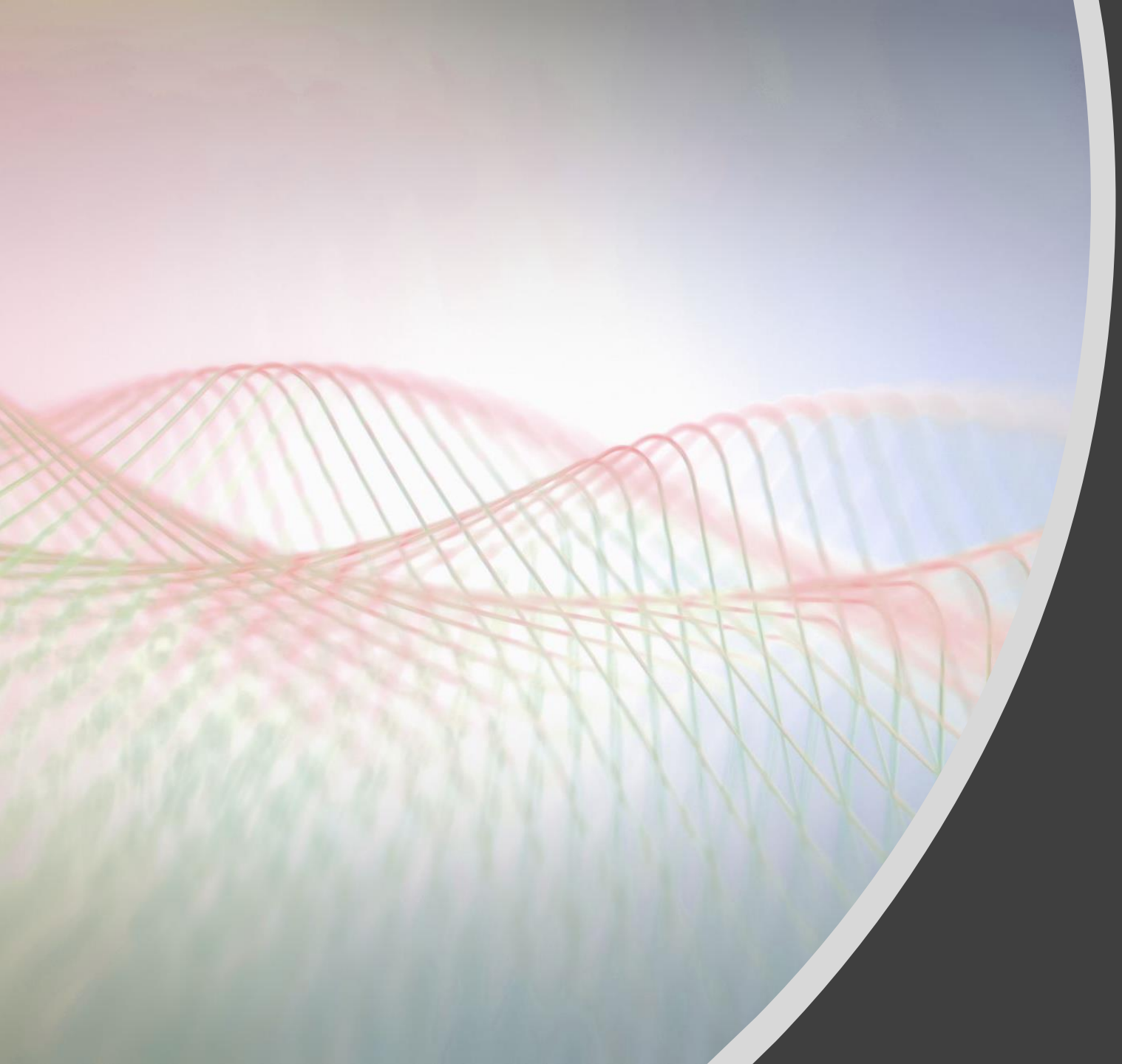


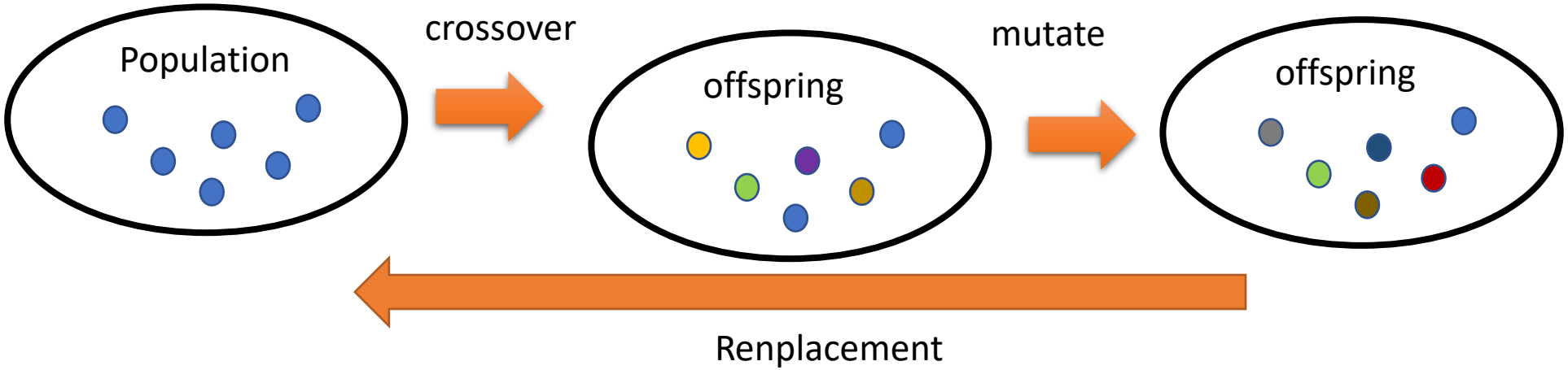
Implémentationd'un algorithme génétique simple avec DEAP

Application sur le problème du OneMax

Schéma général de l'algorithme proposé

Pseudo Code

```
evaluate(population)
for g in range(nbgeneration):
    population = select(population,
len(population))
    offspring = varAnd(population, toolbox, cxpb,
mutpb)
    evaluate(offspring)
    population = offspring
```



Déclaration des paramètres

```
# taille du problème
ONE_MAX_LENGTH = 100
# Paramètres AG
POPULATION_SIZE = 20
P_CROSSOVER = 1.0
P_MUTATION = 1.0
MAX_GENERATIONS = 500
```

Codage du problème de maximisation via la fonction de fitness

```
toolbox = base.Toolbox()
```

toolbox contient en particulier les opérateurs que nous utiliserons.
Nous pouvons l'étendre avec la méthode register

```
# fonction mono objectif qui maximise la première composante de fitness (c'est un tuple)

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

creator est une meta-factory qui permet de créer des classes

La fonction create() est utilisée pour créer une nouvelle classe à partir d'un classe existante en spécialisant des arguments

Utilisation ici de la classe deap.base.Fitness([values]) qui permet de calculer une fitness à partir d'un vecteur de poids (traite donc min et max)

Individus

```
# classe Individual construite avec un conteneur list

creator.create("Individual", list,
fitness=creator.FitnessMax)
```

La classe Individual hérite de list et possède un attribut fitness basé sur notre classe précédente

```
# initialisation des individus avec uniquement des 0

def zero():

    return 0

toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, zero, ONE_MAX_LENGTH)
```

deap.tools.initRepeat(container, func, n) repète func n fois sur le container

Population

```
# initialisation de la population

toolbox.register("populationCreator",
tools.initRepeat, list,
toolbox.individualCreator)
```

Même principe que pour la construction d'un individu mais sur la population entière

```
# Calcul de la fitness/ fonction
toolbox.evaluate()

def oneMaxFitness(individual):

    return sum(individual),  # retourne un tuple
```

On indique ici que la fitness d'un individu est la somme de ses valeurs

```
toolbox.register("evaluate", oneMaxFitness)
```

Définition des opérateurs principaux

```
# Sélection tournoi taille 3
toolbox.register("select",
tools.selTournament, tournsize=3)

# Croisement monopoint
toolbox.register("mate", tools.cxOnePoint)

# Mutation bitflip
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0 / ONE_MAX_LENGTH)
```

On paramètre certains opérateurs génériques disponibles dans la classe tools

Programme principal

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

On initialise la population

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
```

```
stats.register("max", numpy.max)
```

On prepare les statistiques automatiques

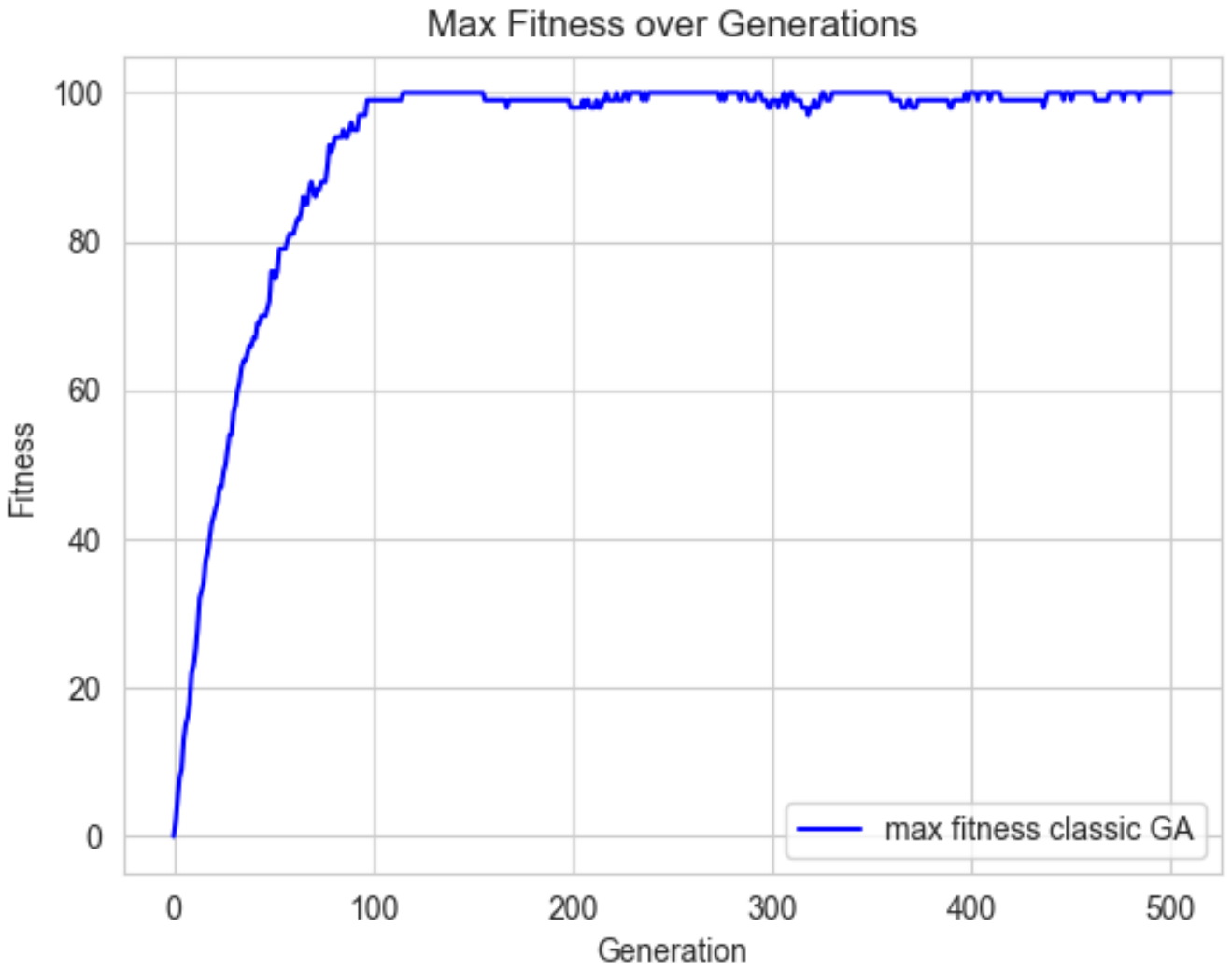
```
population, logbook = algorithms.eaSimple(population, toolbox, P_CROSSOVER, P_MUTATION, MAX_GENERATIONS, stats=stats, verbose=True)
```

Appel de l'algorithme eaSimple avec ses paramètres (il utilisera les opérateurs enregistrés dans la toolbox)

```
maxFitnessValues = logbook.select("max")
```

Récupération des valeurs qui nous intéressent (ici la fitness maximale)

Exécution


Max Fitness over Generations
Fitness
Generation
0
20
40
60
80
100
0
100
200
300
400
500
max fitness classic GA