# Algorytmy ewolucyjne

Paweł Kruczkiewicz, Karol Kusper, Milana Lukasiuk

Projekt czwarty

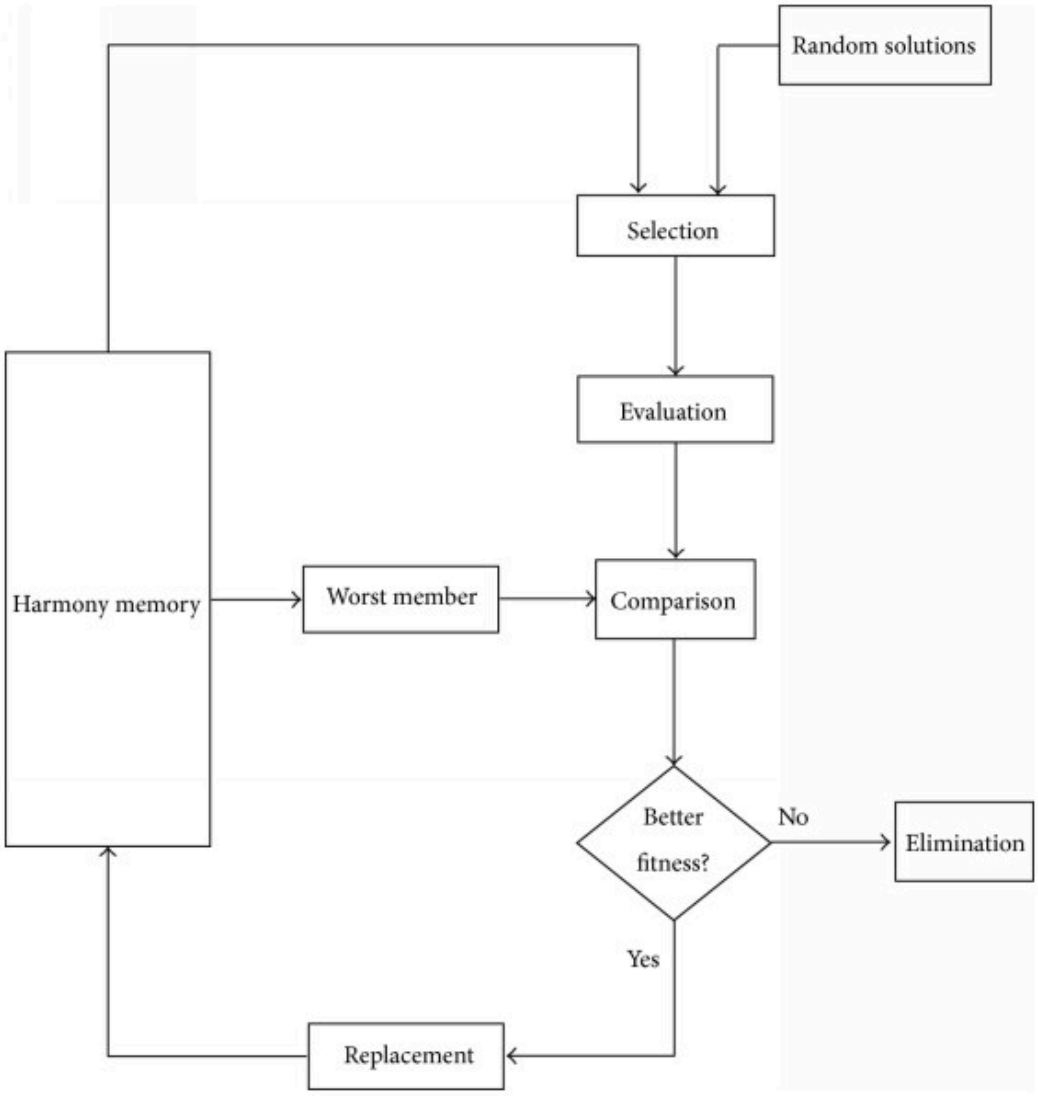
**Optymizacja algorytmem Harmony Search**

*Harmony Search* to meta-heurystyczny algorytm optymalizacyjny opracowany w 2001 roku przez Geem et. al. inspirowany procesem komponeowania muzyki. Wyróżnia go łatowść połączona ze sporą umiejętnością eksploatacji przestrzeni. Został użyty m. in. do optymalizacji funkcji, projektowania struktur mechanicznych, czy minimalizacja kosztów systemów elektroenergetycznych.

Zasada działania algorytmu opiera się o zapamiętywanie najlepszych osobników w tzw. Harmony Memory. Wykorzystuje dwa hiperparametry:

1. `c_r` - Harmony Memory Consideration Rate.
2. `pa_r` - Pitch Adjustment Rate

Przebieg procesu jest przedstawiony na poniższym schemacie:




Kolejne kroki algorytmu to:

1. Inicjalizacja Harmony Memory (HM) losowymi wartościami. Każdy wiersz to potencjalne rozwiązanie.
2. "Improwizacja" nowego rozwiązania: Tworzymy nowe rozwiązanie (*harmonię*). Dla każdego z wymiarów ("nut") nowego rozwiązania:
   A. Z prawdopodobieństwem `c_r` wybieramy losową liczbę z Harmony Memory. W przeciwnym razie bierzemy losową liczbe z zakresu.
   B. Jeżeli nowy składnik jest z HM, to z prawdopodobieństwem `pa_r` "dostrajamy" "nutę", tj. dokonujemy modyfikacji (np. przez dodanie losowej wartości z rozkładu normalnego)
3. Po stworzeniu nowej harmonii (n-wymiarowej listy "nut"), sprawdzamy, czy jest lepsza niż najgorsza harmonia w HM. Jeżeli tak, to zamieniamy.
4. Powtarzamy aż do osiągnięcia warunku stopu.

W niniejszym sprawozdaniu skorzystaliśmy z algorytmu zaimplementowanego w bibliotece mealpy. Skorzystaliśmy z klasy `DevHS`, która implementuje powyższy algorytm z użyciem biblioteki `numpy`.

In [20]:
```
import mealpy
mealpy.__version__
```

Out[20]:
```
'3.0.1'
```

# Konfiguracja, definicja funkcji

Na początku importujemy potrzebne zależności, implementujemy funkcję fitness oraz definijemy "problem dict", tj parametry przekazywane do solvera z MealPy.

In [2]:
```
from mealpy import FloatVar, HS
from pathlib import Path
import matplotlib.pyplot as plt
from time import time


MIN_RANGE, MAX_RANGE = -65.536, 65.535
N_DIMS = 2
EPOCHS = 50
POP_SIZE = 50
SEED = 42

RESULTS_DIR = Path("results")

def elipsoid_function(x):
    n = len(x)
    return sum(x[i]**2 for j in range(n) for i in range(j))

PROBLEM_DICT = {
    "obj_func": elipsoid_function,
    "bounds": FloatVar(lb=(MIN_RANGE, ) * N_DIMS, ub=(MAX_RANGE,) * N_DIM
    "minmax": "min",
    "name": "Harmony Search Default",
    "log_to": "file",
    "log_file": RESULTS_DIR / "harmony_search.txt"
}
```

Sprawdzamy, czy wywołanie funkcji z parametrami domyślnymi pozwala zoptymalizować zadaną funkcję.

In [3]:
```
c_r_0, pa_r_0 = 0.15, 0.5

optimizer = HS.DevHS(epoch=EPOCHS, pop_size=POP_SIZE, c_r=c_r_0, pa_r=pa_
g_best = optimizer.solve(PROBLEM_DICT, seed=SEED)
```

Optymalizacja działa! Możemy sprawdzić, czy wynik jest zgodny z oczekiwaniami.

Zapis logów z powyższej optymalizacji powinien być zapisany w `results/HS Harmony Search Defaults.txt`.

In [4]:
```
print(f"Solution = {g_best.solution}, Fitness: {g_best.target.fitness}")
```

```
Solution = [ 1.10163092e-02 -2.45786380e+01], Fitness: 0.00012135906944478
201
```

MealPy pozwala na zobaczenie całej historii procesu optymalizacji przez epoki. Warto nadmienić, że obiekt `History` posiada również sporo przydatnych funkcji, tj. np. możliwość wygenerowania i zapisania wykresu najlepszego fitness w zależności od epoki.

In [21]:
```
optimizer.history.list_global_best_fit[:10]
```

Out[21]:
```
[np.float64(9.689665419186671),
 np.float64(2.786380638932241),
 np.float64(0.8205635500868906),
 np.float64(0.7011079507863216),
 np.float64(0.27211410188558804),
 np.float64(0.03022895399060911),
 np.float64(0.03022895399060911),
 np.float64(0.0015692145877577333),
 np.float64(0.0015692145877577333),
 np.float64(0.0015692145877577333)]
```

## Stworzenie funkcji

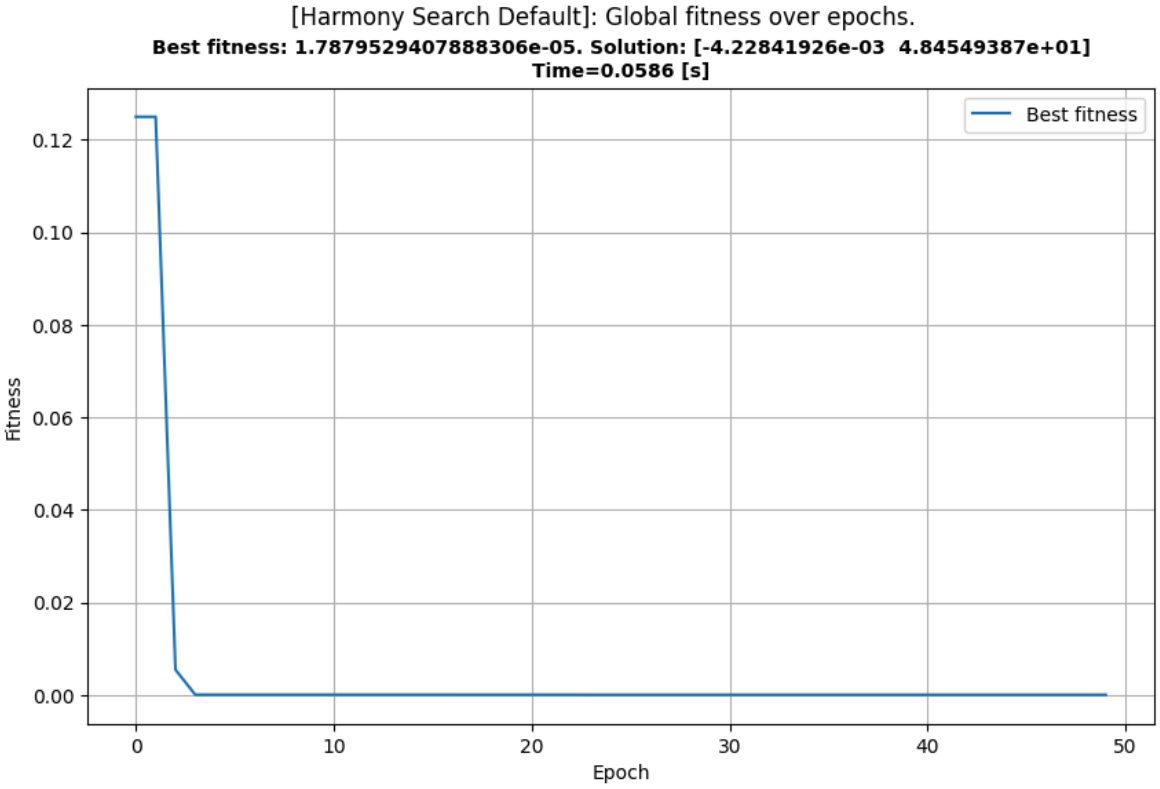
Po sprawdzeniu, ze MealPy działa, przechodzimy do stworzenia fukcji pomocniczych.

Zadaniem funkcji `optimize_HS` jest przybliżenie interfejsu udostępnianego przez MealPy do interfejsu z poprzednich projektów. Przyjmuje ona jedynie hiperparametry oraz, dla rozróżnalności, nazwę danego optymalizatora. Zwraca słownik z nazwą, rozwiązaniem, najlepszym fitness , listą najlepszych globalnych fitness w każdej epoce, oraz czas wykonania.

`plot_fitness_history` to funkcja, która na bazie zwracanego słownika, tworzy wykres przedstawiający wszystkie wyniki.

In [16]:
```
def optimize_HS(c_r, pa_r, name="Harmony Search"):
    problem_dict = {**PROBLEM_DICT, "name": name, "log_file": RESULTS_DIR

    optimizer = HS.DevHS(epoch=EPOCHS, pop_size=POP_SIZE, c_r=c_r, pa_r=p
    t1 = time()
    g_best = optimizer.solve(problem=problem_dict)
    total_time = time() - t1

    # optimizer.history.save_global_best_fitness_chart(filename=(RESULTS_

    return {"name": name,
            "solution": g_best.solution,
            "fitness": g_best.target.fitness,
            "history_fitness": optimizer.history.list_global_best_fit,
            "time": total_time}


def plot_fitness_history(g_best_dict,):
    fitness_list, solution, fitness, name, time = g_best_dict["history_fi
    epochs = list(range(EPOCHS))

    plt.figure(figsize=(10, 6))
    plt.plot(epochs, fitness_list, label='Best fitness')
```

```
    plt.suptitle(f"[{name}]: Global fitness over epochs.")
    plt.title(f"Best fitness: {fitness}. Solution: {solution}\nTime={time
    plt.xlabel("Epoch")
    plt.ylabel("Fitness")
    plt.legend()
    plt.grid(True)
    plt.savefig((RESULTS_DIR/f"HS_{name}.png").as_posix())
    plt.show()


g_best = optimize_HS(c_r=c_r_0, pa_r=pa_r_0, name="Harmony Search Default
plot_fitness_history(g_best)
```



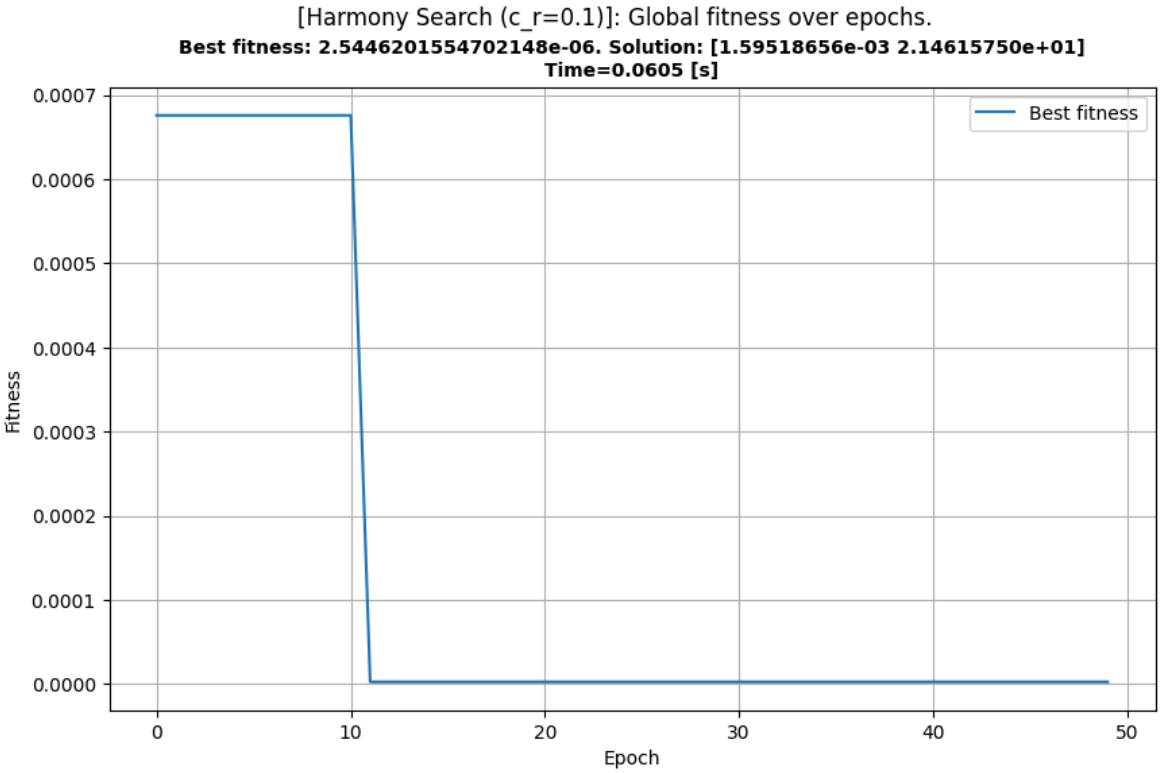

# Testy z różnymi hiperparemterami

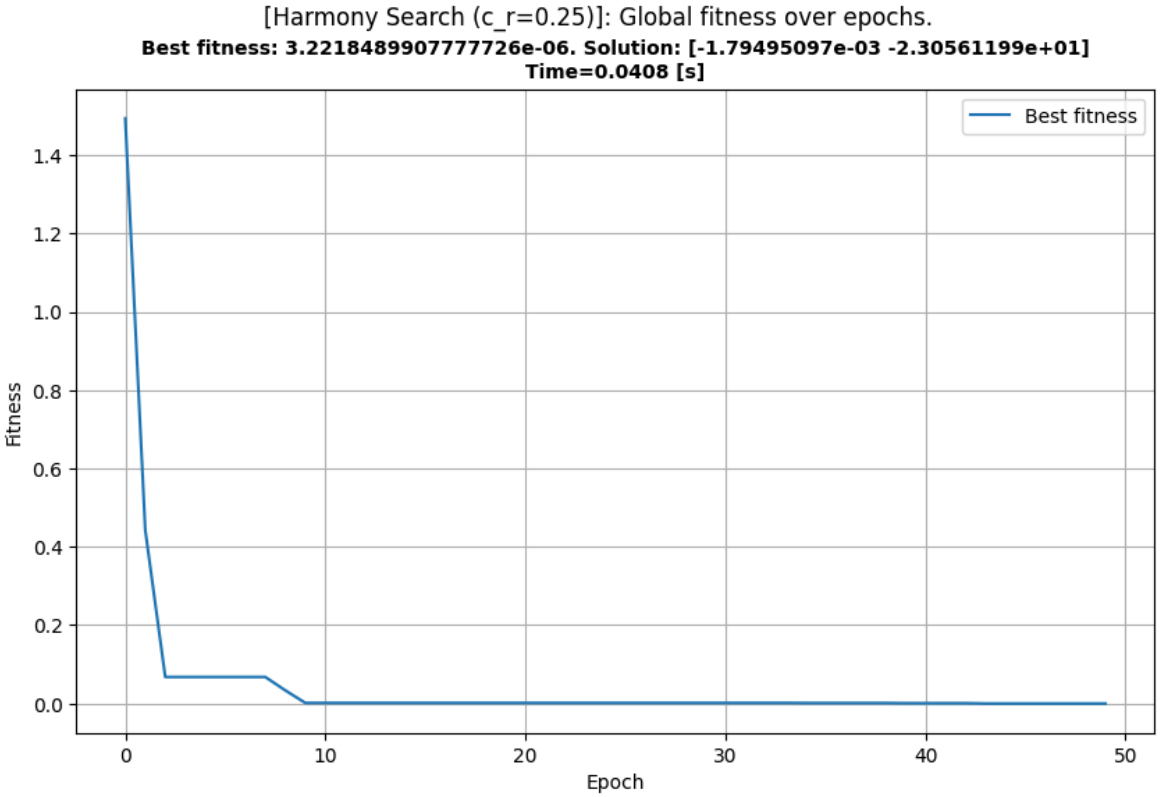
W kolejnej części sprawdzamy różne hiperparametry. Porównanie następuje poprzez sprawdzenie wykresów dla różnych wartości hiperparametrów.

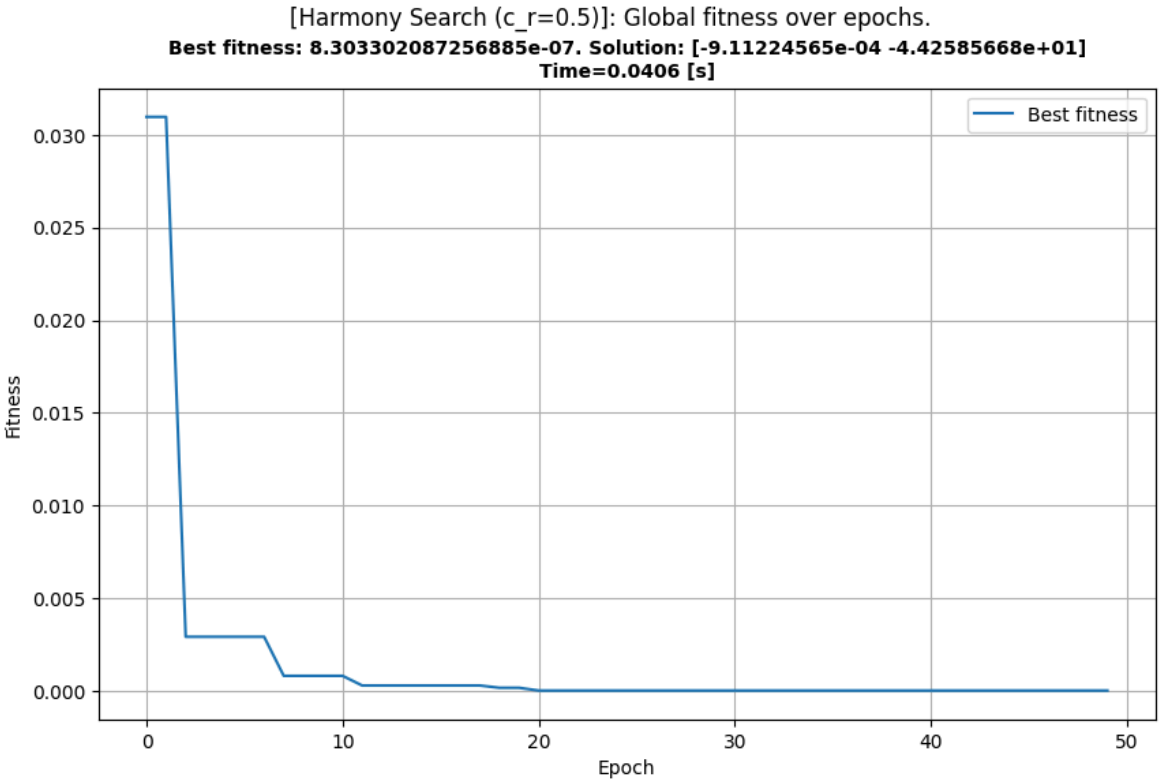
## Harmony Memory Consideration Rate ( `c_r` )

Parametr Harmony Memory Consideration Rate to prawdopodobieństwo wybrania nuty z Harmony Memory. Dla `c_r -> 0`, algorytm jest losowym przeszukiwaniem z pamięcią.

In [17]:
```
c_rs = [0.1, 0.25, 0.5]

for c_r in c_rs:
    g_best = optimize_HS(c_r=c_r, pa_r=pa_r_0, name=f"Harmony Search (c_r
    plot_fitness_history(g_best)
```


[Harmony Search (c_r=0.1)]: Global fitness over epochs.
Best fitness: 2.5446201554702148e-06. Solution: [1.59518656e-03 2.14615750e+01]
Time=0.0605 [s]
Best fitness
Fitness
Epoch
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
10
20
30
40
50


[Harmony Search (c_r=0.25)]: Global fitness over epochs.
Best fitness: 3.2218489907777726e-06. Solution: [-1.79495097e-03 -2.30561199e+01]
Time=0.0408 [s]
Best fitness
Fitness
Epoch
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50

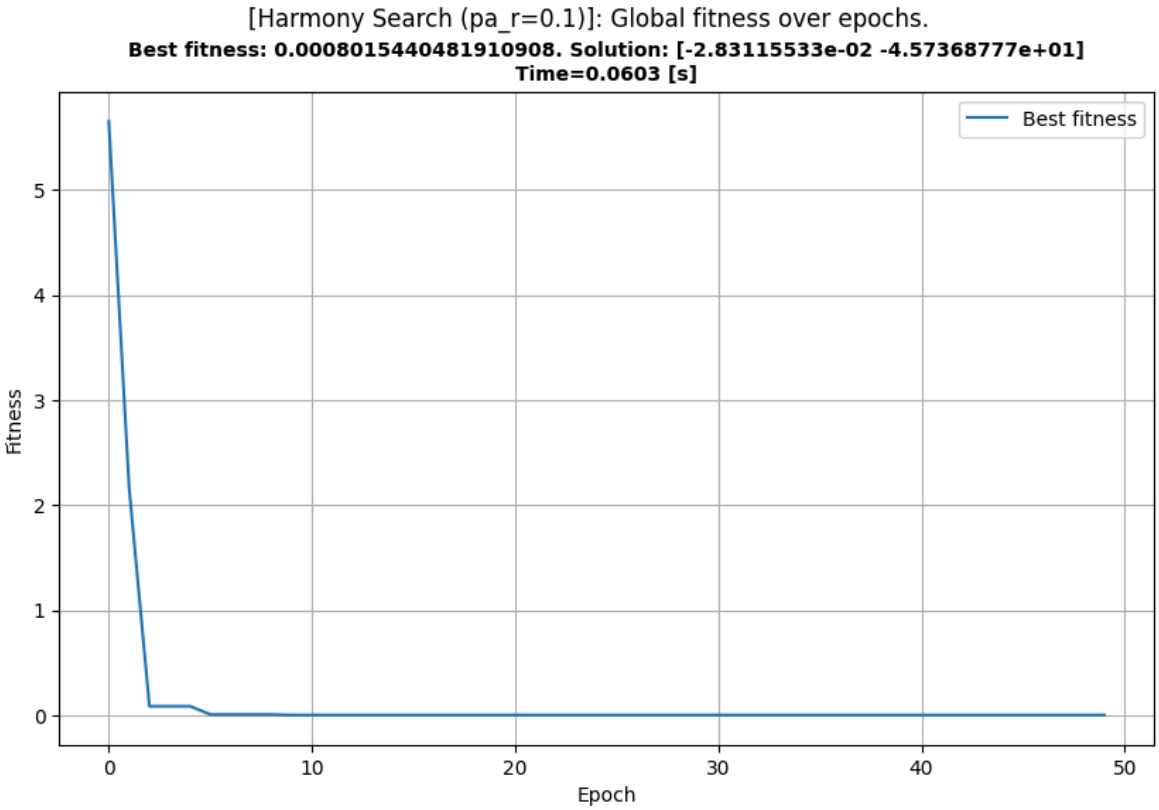


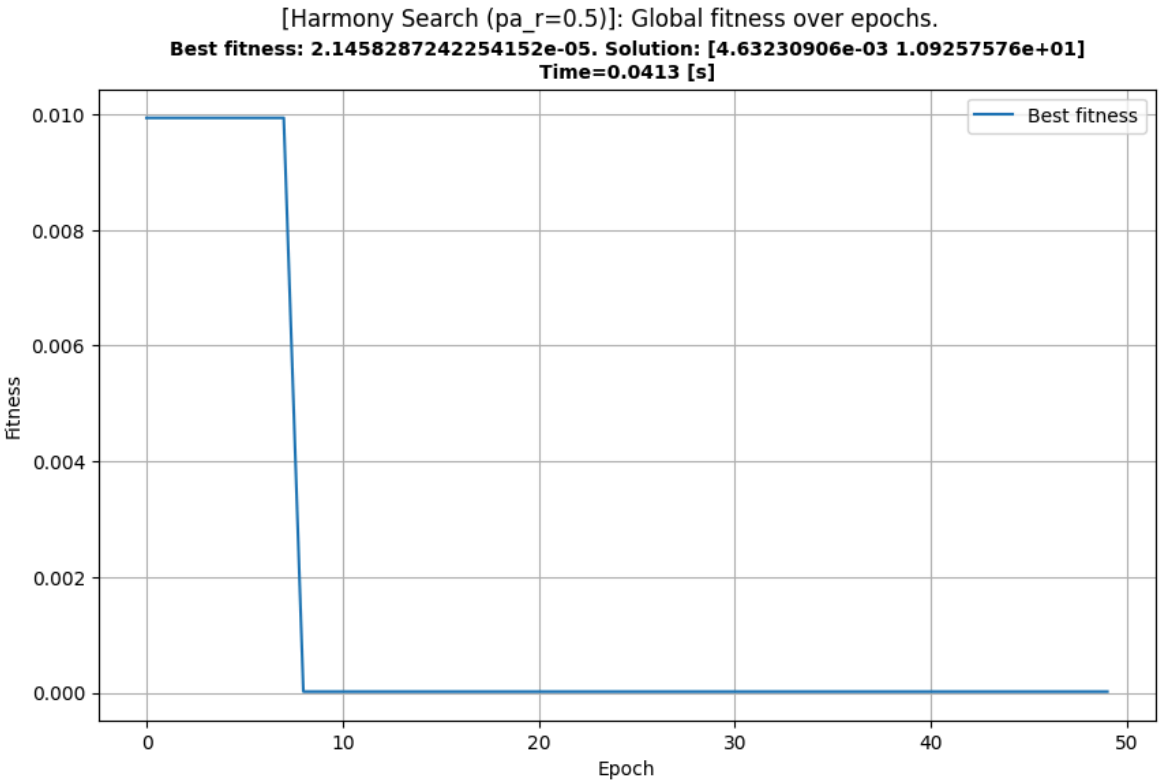

Można zauważyć, że algorytm o `c_r` wyższym uzyskał wynik lepszy niż dla mniejszych wartości. Wskazuje to zatem, że zapisywane i pobieranie wartości w HM pozwala na lepsze zbieganie algorytmu.

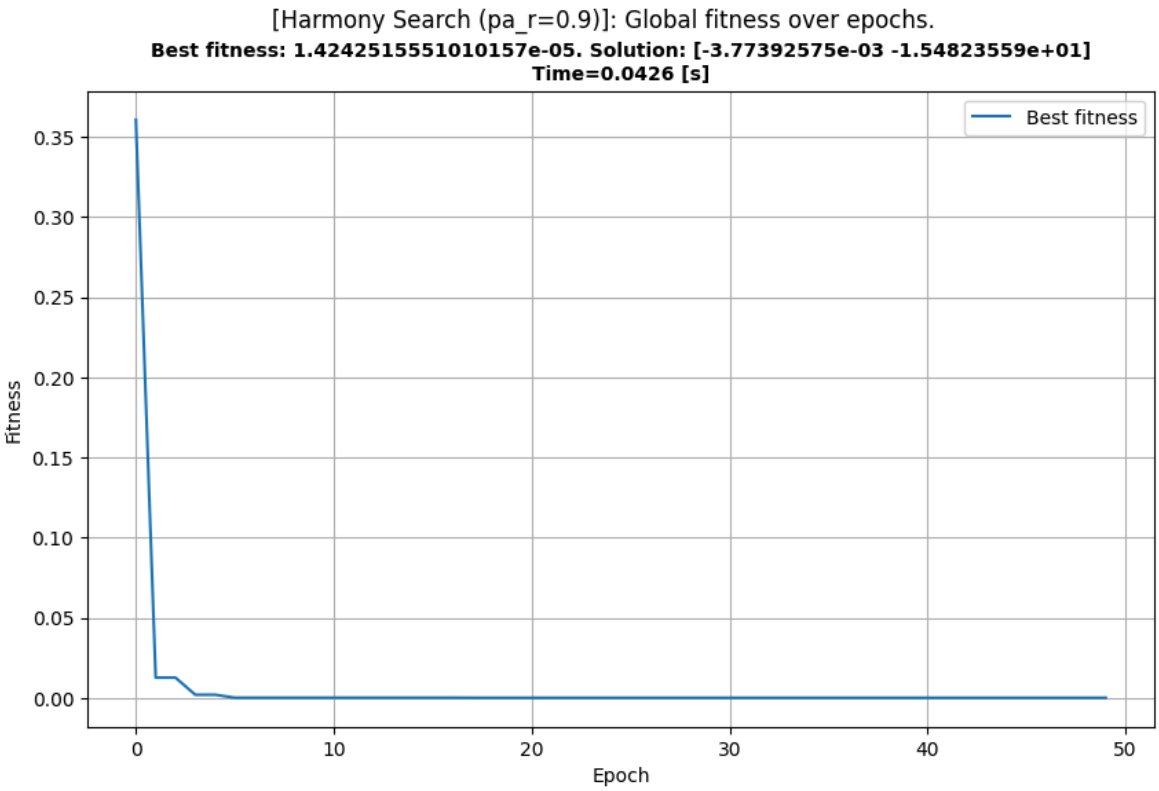
## Pitch Adjustment Rate ( `pa_r` )

Pitch Adjustment Rate to prawdopodobieństwo zastosowania zmiany wartości dla nowej "nuty". Jest to analogiczne do mutacji w algorytmach ewolucyjnych.

In [19]:
```
pa_rs = [0.1, 0.5, 0.9]

for pa_r in pa_rs:
    g_best = optimize_HS(c_r=c_r_0, pa_r=pa_r, name=f"Harmony Search (pa_
    plot_fitness_history(g_best)
```


[Harmony Search (pa_r=0.1)]: Global fitness over epochs.
Best fitness: 0.0008015440481910908. Solution: [-2.83115533e-02 -4.57368777e+01]
Time=0.0603 [s]
Best fitness
Fitness
Epoch
0
1
2
3
4
5
0
10
20
30
40
50


[Harmony Search (pa_r=0.5)]: Global fitness over epochs.
Best fitness: 2.1458287242254152e-05. Solution: [4.63230906e-03 1.09257576e+01]
Time=0.0413 [s]
Best fitness
Fitness
Epoch
0.000
0.002
0.004
0.006
0.008
0.010
0
10
20
30
40
50




Wartości 0.5 oraz 0.9 algorytmu poradziły sobie nieco lepiej niż 0.1, które zasadniczo pozbawia możliwości mutacji (a zatem zmniejsza eksplorację na rzecz eksploatacji). Można zatem stwierdzić, że warto użyć tego algorytmu do eksploatacji.

# Podsumowanie

Algorytm Harmony Search wykazuje duże podobieństwo do algorytmu ewolucyjnego opartego o rzeczywistą reprezentację chromosomu. Zarówno schemat działania, jak i osiągnięte wyniki to potwierdzają. Oba algorytmy znajdują satysfakcjonujące minimum funkcji hiperelipsoidy zazwyczaj w mniej niż 10 epok. Jest to zadowalający wynik.

Warto jednak zauważyć, że algorytm ewolucyjny przedstawiony w sprawozdaniu 2 osiąga wynik nieco bliższy matematycznie wyznaczonemu minimum elipsoidy. Może to być związane z większą liczbą "ulepszeń" (jak na przykład osobniki elitarne, bardziej zaawansowany crossover, itd.), lecz również może być to skutek zabezpieczeń przy operacjach na liczbach zmiennoprzecinkowych, które aplikuje biblioteka MealPy.

Czasy działania obydwu algorytmów są podobne. Przy tak małym i prostym problemie, różnice kilku setnych sekundy nie mają większego znaczenia, a obliczanie procentu przewagi może dawać mylne wyniki ze względu na błąd statystyczny. Warto również zaznaczyć, że biblioteka MealPy udostępnia mozliwość sprawdzenia czasu każdej iteracji, co może być pomocne przy debugowaniu kodu.

Ze względu na reprezentację rzeczywistą, trudno porównać obecny algorytm z tym przedstawionym w zadaniu 1.

Kończąc, biblioteka MealPy daje spore możliwości nie tylko na zapoznanie się z, ale także na wykorzystanie ponad 200 algorytmów optymalizacyjnych. Daje to spore mozliwości rozwoju, a spójność interfejsów pomiędzy różnymi algorytmami sprawia, że można łatwo podmieniać algorytmy w zastosowaniach. Poznany algorytm Harmony Search przyniósł wyniki zbliżone do klasycznego algorytmu ewolucyjnego dla wszystkich zastosowanych parametrów.