Sprawozdanie

Ćw. 8 – Symulowane wyżarzanie

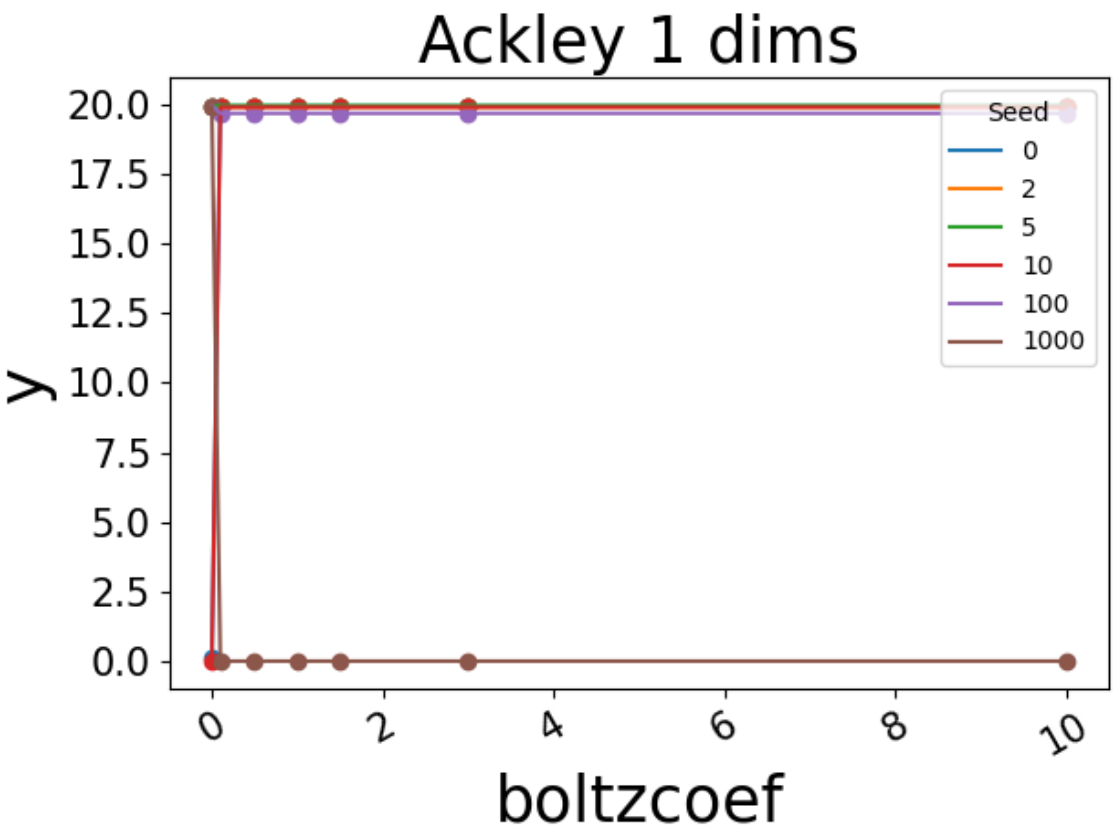
Filip Horst 311257

1 Badanie implementacji C++

Badania odbywały się poprzez zmianę tylko jednego parametru na raz w celu sprawdzenia, czy występują jakieś zależności. Wnioski i obserwacje były tworzone na podstawie wykresów oraz ręcznego przeglądu plików wyjściowych. Niektóre badania zostały opisane w dużym skrócie, jeśli nie wniosły nic wartościowego i/lub pokrywały się z innym przypadkiem. Badane zakresy wykorzystane do porównań były takie same dla wszystkich funkcji, jednak w niektórych szczegółowych badaniach były zmienianie w celu przeszukania dokładniejszych zakresów i analizy cech algorytmu.

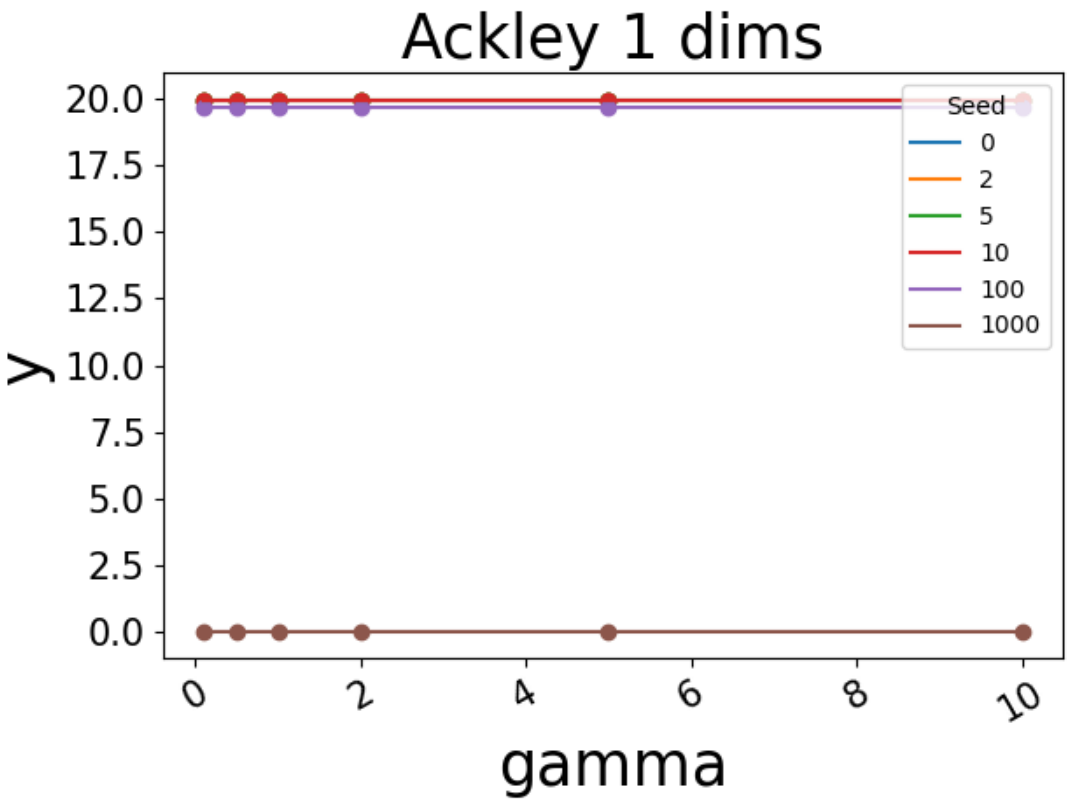
1.1 Funkcja Ackley'a

1.1.1 1 wymiar



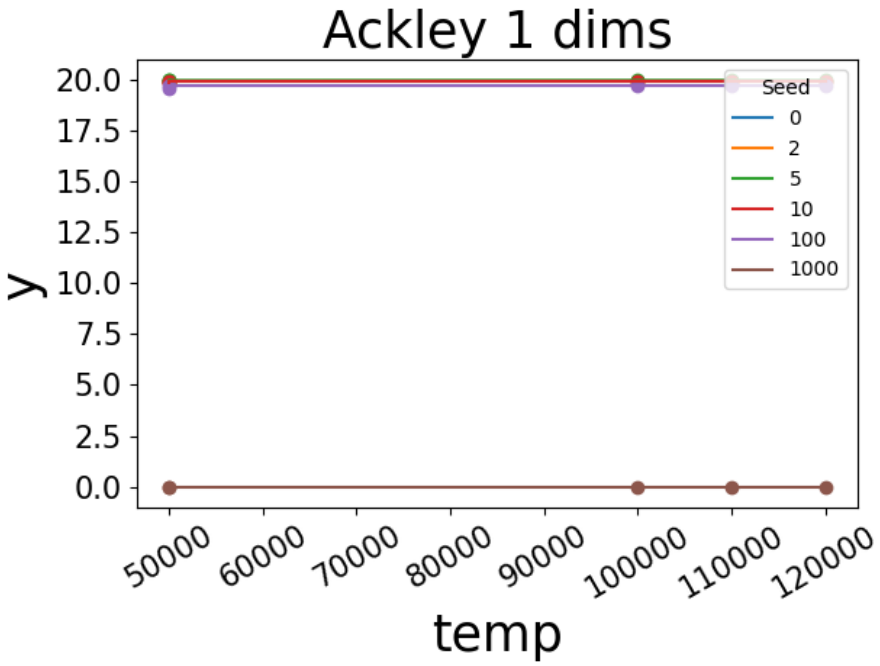
Zależność wyniku y od ustawień współczynnika Boltzmana

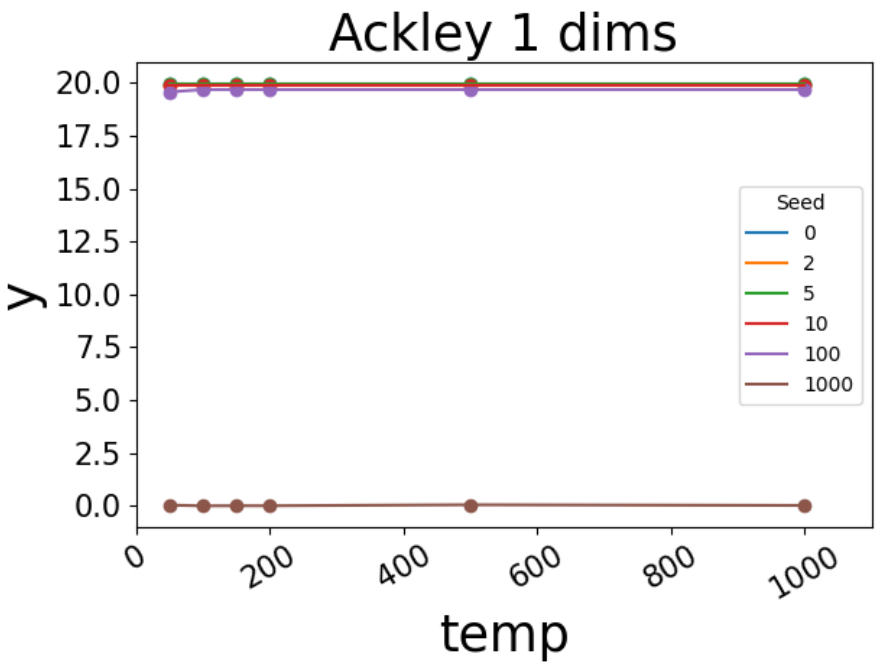
Wpływ ustawień wsp. Boltzmana na wyniki był niezauważalny – wyniki pozostawały stałe dla każdego ustalonego ziarna. Wyjątkiem było ustawienie kb = 0, gdzie wyniki się zmieniły, jednak nie było w nich wyraźnej zależności. Dla większości kb =0 pogorszyło wyniki, ale np. dla seed = 10 wynik był wtedy lepszy.



Zależność wyniku od ustawień gamma

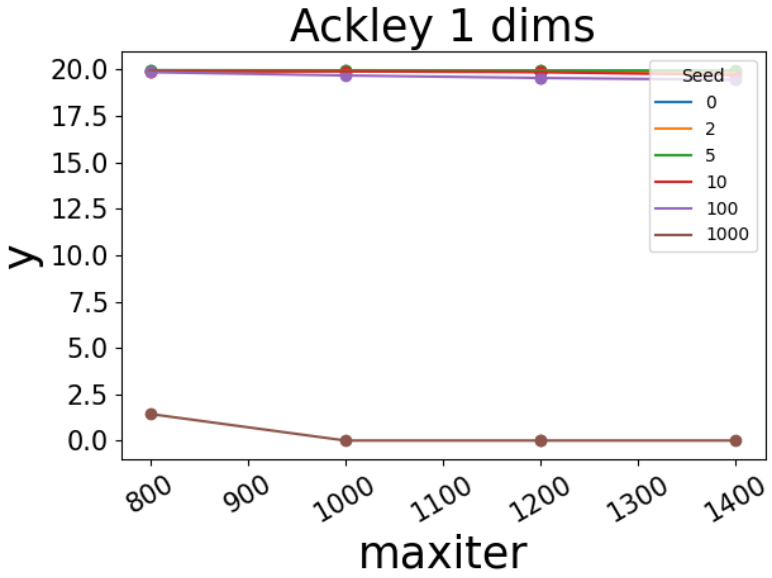
Gamma w odróżnieniu od stałej Boltzmana nie miała absolutnie żadnego wpływu na wynik niezależnie od testowanych ustawień.





Zależność wyniku y od ustawień temperatury – dwa zakresy badań przedstawione na dwóch wykresach

Również temperatura nie ma zauważalnego wpływu niezależnie od badanego zakresu.

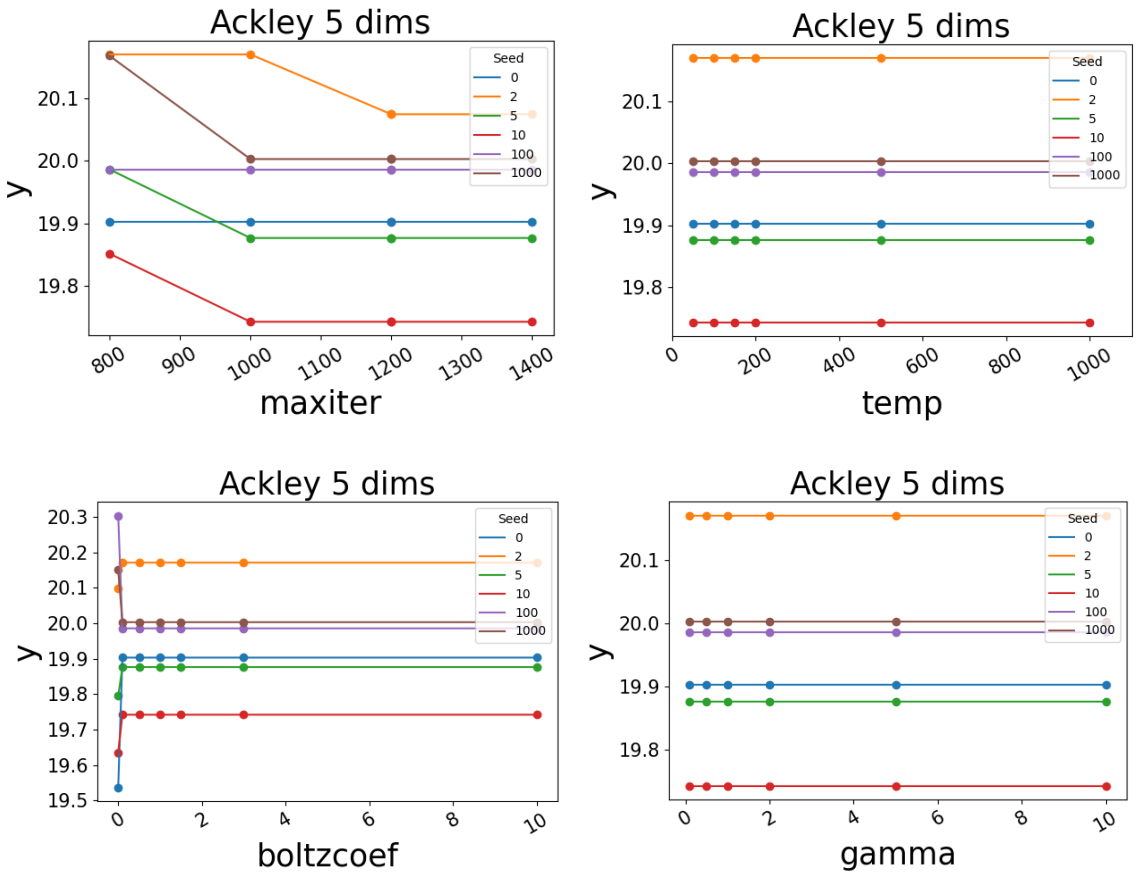


Zależność wyniku od ustawień maxiter

Najbardziej wyraźną zależność miał parametr maxiter, ponieważ wynik końcowy poprawiał się wraz ze zwiększaniem iteracji. Dla części seed widać jednak, że po pewnym momencie wynik się zatrzymywał i większa liczba iteracji nic nie dała. Można to jednak porównać do zwykłego zgadywania – oczywiście im więcej „strzałów” wykonamy tym większa szansa, że coś się uda. Istnieje również szansa, że w badanym przypadku wynik wpadał w minimum lokalne, a ze względu na zaimplementowaną metodę tworzenia kolejnego rozwiązania przez dodanie szumu nie był on w stanie się z niego wybić, więc niezależnie od liczby iteracji algorytm tkwił w miejscu.

Ogólnie należy również wspomnieć, że tylko jedno rozwiązanie było bardzo dobre, bo trafiło już na samym początku w okolice optimum.

1.1.2 5 wymiarów

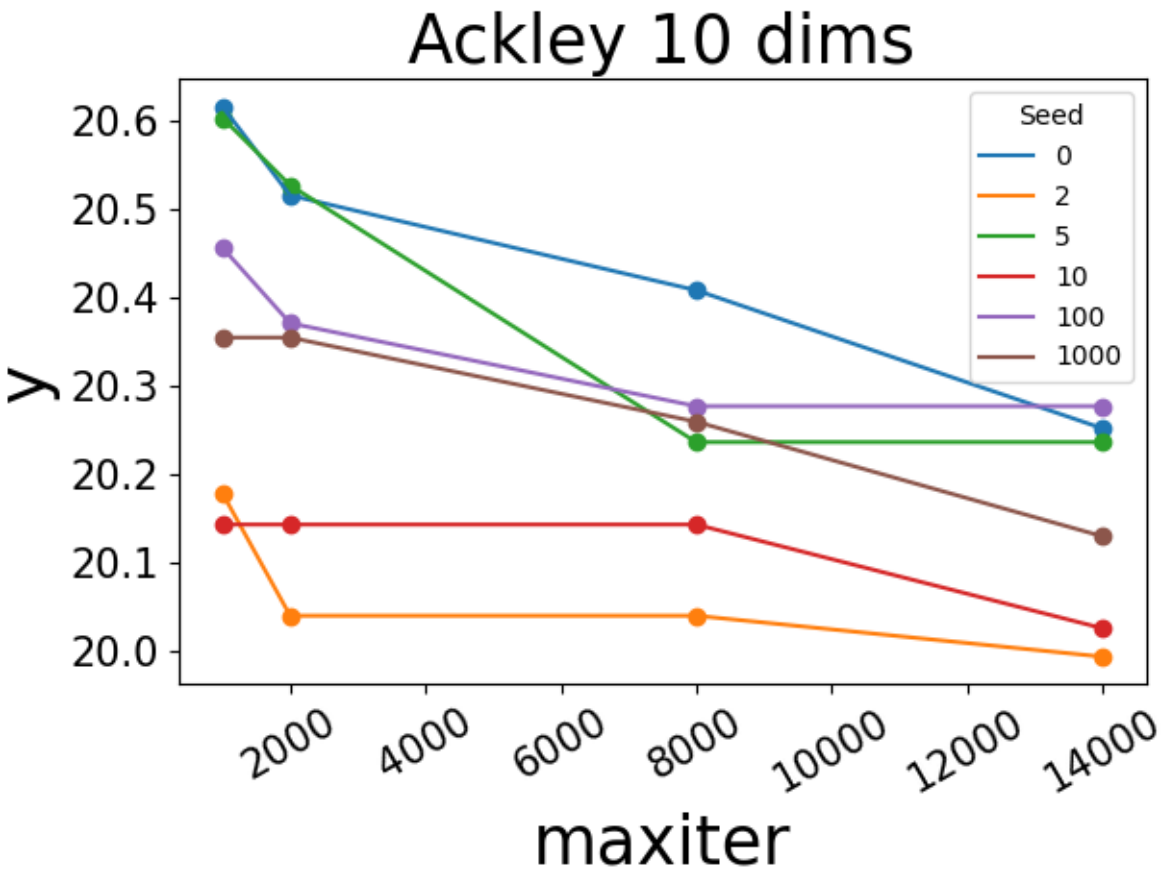


Dla 5 wymiarów większość zależności (a raczej ich brak) był analogiczny.

Jak można zaobserwować po samych wartościach osi Y, algorytm nie poradził sobie zbyt dobrze z rozwiązaniem problemu dla 5 wymiarów.

1.1.3 10 wymiarów

Dla 10 wymiarów podobnie jak w przypadku 5 brakowało jakichkolwiek zależności. Jednak z powodu wyglądu wykresu Y od maxiter zdecydowano się na zbadanie dużo większej liczby iteracji niż poprzednio, by sprawdzić czy ich zbyt mała ilość może być powodem słabych wyników.



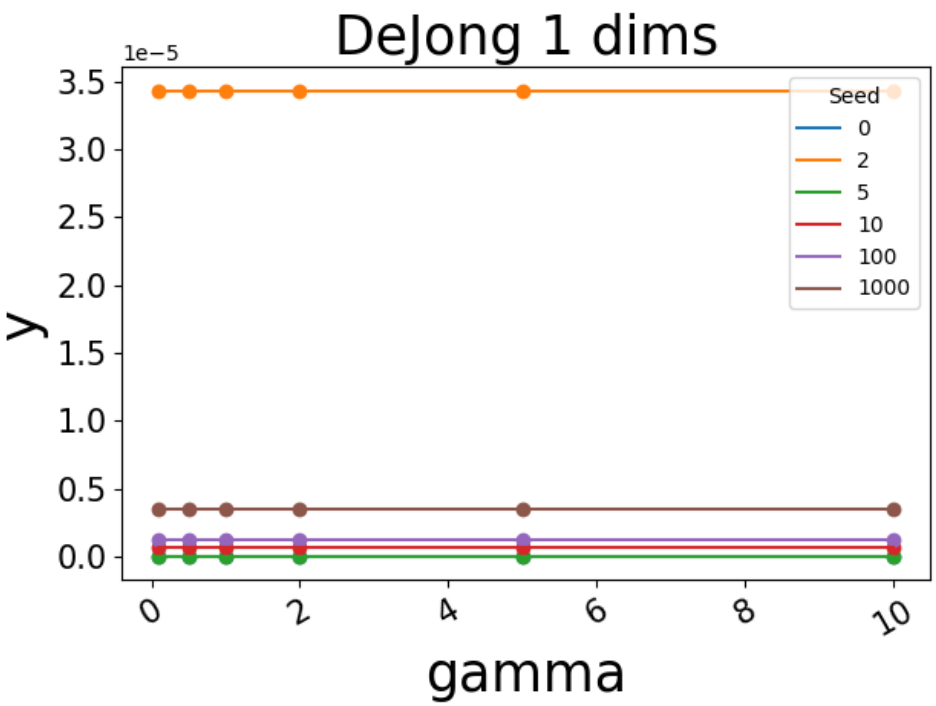
Zależność Y od maxiter

Jako przykład jednego parametru wstawiony został wykres Y od maxiter. Można z niego wywnioskować, że algorytm nie poradził sobie z zadaniem minimalizacji, ponieważ nawet przy 14000 iteracji nie jest w stanie nawet przybliżyć się do 0. W tym badaniu ymin = 19.99356, ale do porównania wybrany został przypadek dla 1400 iteracji gdzie ymin = 20.14332, żeby zapewnić zgodność z ustawieniami w pozostałych przypadkach. Widać jednak, że dodatkowe iteracje trochę pomogły.

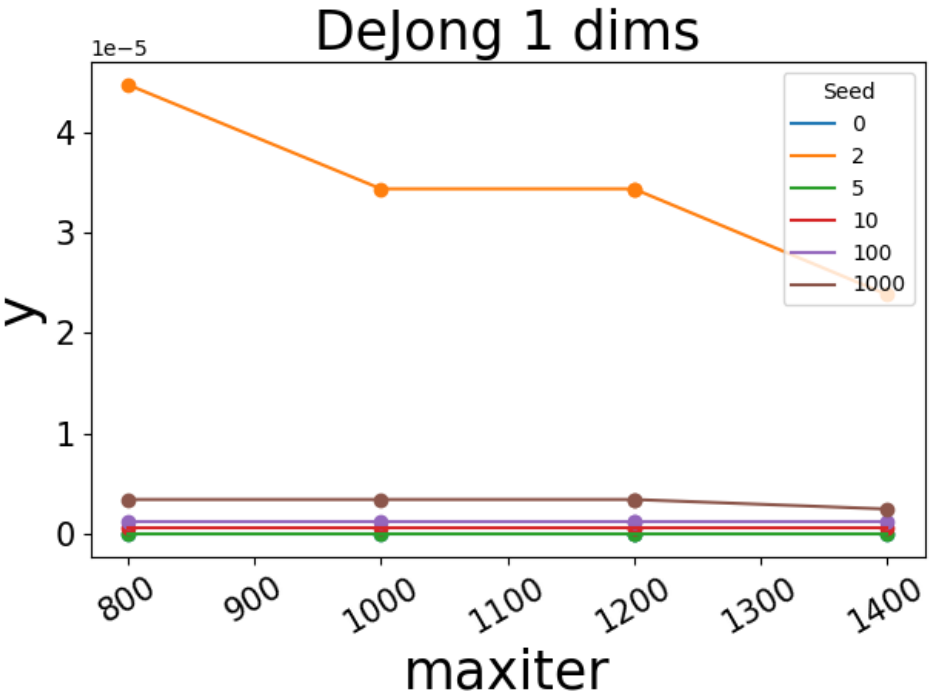
1.2 Funkcja DeJonga

1.2.1 1 wymiar

Podobnie jak w funkcji Ackley'a nie udało się znaleźć żadnej zależności wyników od parametrów poza maxiter. Załączony przykładowy parametr gamma, który przedstawia prostą poziomą:



Wynik w zależności od Gamma



Wynik w zależności od iteracji

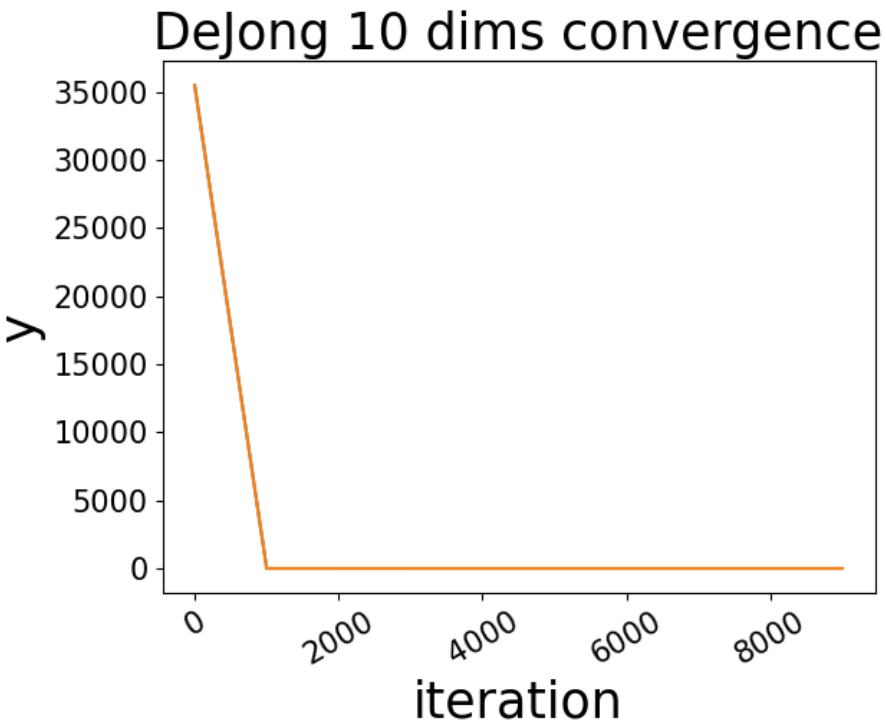
Można jednak zauważyć, że wyniki są bliskie oczekiwanemu, a w niektórych przypadkach prawie równe. Można podejrzewać, że wyniki dla gorszych seed można by było poprawiać dalej zwiększając liczbę iteracji.

1.2.2 5 wymiarów

Podobnie jak w przypadku funkcji Ackley'a dla większej liczby wymiarów nie udało się osiągnąć poprawnych wyników. Najlepszy okazał się y = 0.04234984, co jest dość dobrym wynikiem, ale w zależności od problemu może być uznany za daleki od optimum.

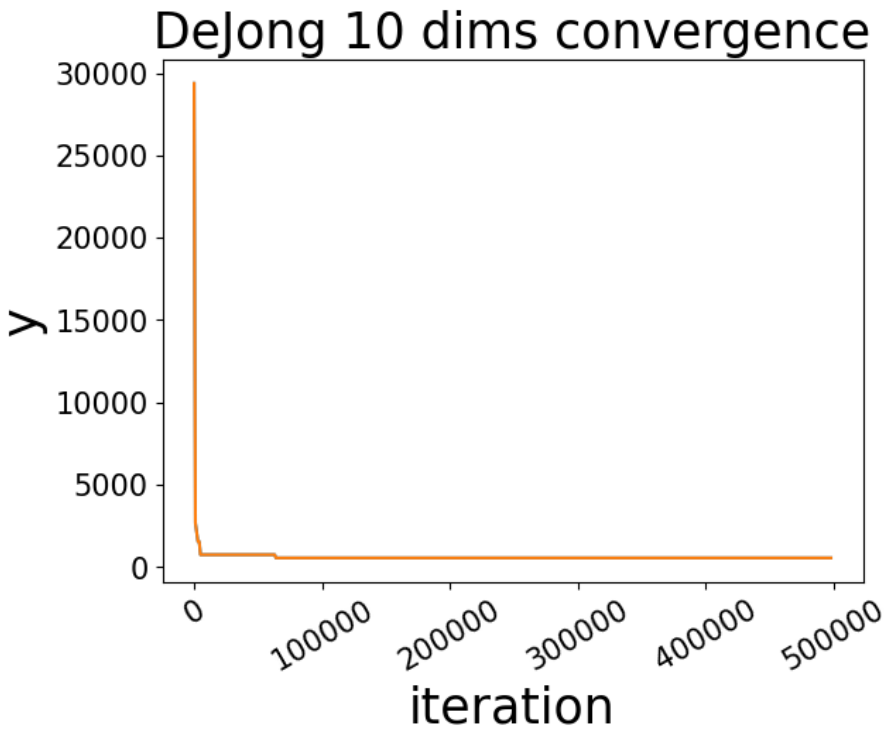
1.2.3 10 wymiarów

Dla 10 wymiarów przy ustalonych 1400 iteracjach udało się uzyskać wynik zaledwie y = 0.5510613. W tym przypadku przeprowadzono dodatkowe doświadczenie polegające na sprawdzeniu zbieżności dla 10 000 iteracji. Wybrane zostało inne jądro generatora, by sprawdzić czy dzięki większej liczbie zdoła „dogonić" to najlepsze. Nie udało się to i wynik ostateczny wyniósł y = 0.629611.



Wykres nauki dla 10k iteracji

Wykres pokazuje, że zbyt duża liczba iteracji jest bez sensu. Już w okolicach 1000 iteracji znalezione zostało rozwiązane i od tamtej pory nic się nie zmieniło. Co do wcześniejszych iteracji nie widać żadnych stabilnych i konsekwentnych popraw tylko nagły skok do lepszej wartości. Może to oznaczać zbyt duże zmiany między rozwiązaniami. Żeby to sprawdzić, zmieniono operację tworzenia nowych rozwiązań na dodawanie szumu losowego z przedziału -50, 50 na -1,1.



Przebieg poszukiwań dla szumu -1,1 500k iteracji

Przy takiej wersji algorytmu wymagana liczba iteracji znacznie wzrosła, a wynik się pogorszył, bo wyniósł aż y = 557.4477. Istnieje jednak szansa, że zmiana z szumu +-50 na +-1 była zbyt duża. W ramach eksperymentu zmieniona została również temperatura startowa na T = 1 000 000, ale wynik się nie zmienił. Dla porównania dla szumu +- 50 i 500 000 iteracji y wyniósł y = 0.0007154978, co jest znaczną poprawą.

1.3 Funkcja Rastragina

1.3.1 1 wymiar

Wszystkie obserwacje są analogiczne do poprzednich funkcji. Najlepszy wynik przy ustalonych 1400 iteracjach max y = 0.002085363.

1.3.2 5 wymiarów

Najlepszy wynik wyniósł zaledwie y = 9.256569.

1.3.3 10 wymiarów

Nic się nie zmieniło. W celach porównawczych najlepszy wynik y = 64.21758 został zapisany przy ustawieniu 1400 iteracji.

2 Porównanie

Wykonane dla ustawień, gdzie maksymalna liczba iteracji wynosiła 1400. Reszta parametrów pozostała w ustawieniach domyślnych T = 100 000, kB = 1, gamma = 1, ponieważ nic nie wskazywało na to, że mają jakieś poważne znaczenie. Tabela ma na celu porównanie jakie funkcje są najtrudniejsze dla algorytmu.

Funkcja	Wymiary	Najlepszy wynik
Ackley	1	0.002481523
Ackley	5	19.53637

Ackley	10	20.14332
DeJong	1	3.725518e-09
DeJong	5	0.04234984
DeJong	10	0.5510613
Rastragin	1	1.667147e-06
Rastragin	5	9.256569
Rastragin	10	64.21758

Według osiąganych wyników można powiedzieć, że najtrudniejszą funkcją dla tego algorytmu jest f. Rastragina, średnio trudna jest f. Ackley'a, a najprostsza (wciąż raczej trudna) jest funkcja DeJong'a.

2.1 Wnioski

Na podstawie badań powstają następujące wnioski:

- Parametry mają niewielki wpływ na ostateczne wyniki algorytmu
- Największy wpływ ma czynnik losowy, czyli seed oraz liczba iteracji
- Algorytm radzi sobie bardzo dobrze dla funkcji o bardzo niskiej liczbie wymiarów (łatwych)
- Zbieżność algorytmu jest nieprzewidywalna – brakuje stabilnych i konsekwentnych popraw wraz z iteracjami. Zamiast tego występuje „pauza", a po kilkuset/kilku tysiącach iteracji nagła zmiana – wskazuje to na losowość w działaniu.
- Optymalizacja trudniejszych funkcji daje się wykonać tylko częściowo dla bardzo dużej liczby iteracji – nie da się ocenić, czy dalsze ich zwiększanie doprowadziło by do optimum
- Trudność funkcji ma wpływ na osiągane wyniki
- Algorytm działa szybko – dla większości testowanych ustawień ok. 1 sekundy lub nawet mniej
- Duży wpływ losowości znacznie utrudniał analizę potencjalnego niskiego, ale nie zerowego wpływu innych parametrów na wyniki. Efektywna wizualizacja oraz ręczny odczyt i porównanie z raportów był bardzo trudny – praktycznie niemożliwy

Podsumowując, prymitywność i prostota algorytmu sprawia, że działa na zasadzie „albo da dobry wynik albo nie". Zauważalny jest brak wpływu innych parametrów niż liczba iteracji, co sprawia że algorytm można uznać bardziej za ulepszoną zgadywankę. Nie można mu jednak odmówić tego, że przy odpowiedniej liczbie prób jest w stanie znajdywać rozwiązania stosunkowo bliskie optimum. Trzeba jednak pamiętać, że „stosunkowo bliskie" według osobistej oceny w prostym badaniu projektowym, a rozwiązanie, które było by wystarczające by rozwiązać nim realny problem to dwie odległe rzeczy. Tak zaimplementowany algorytm jest **użyteczny tylko dla prostych problemów**.

Zaletą SA jest jednak błyskawiczne działanie, co pozwala na użycie go jako wstępu do bardziej zaawansowanego algorytmu. Można wykonać bardzo szybkie kilka tysięcy iteracji S, by przybliżyć się w kierunku rozwiązania, a następnie wystartować ze znalezionego punktu bardziej wyszukanym i potencjalnie kosztowniejszym innym algorytmem. Takie podejście było by szczególnie skuteczne dla f. DeJonga, bo w tym przypadku SA dochodziło do wartości bliskich optimum.

3 Dodatkowe badanie – zaawansowane SA w Python

Badanie polegało na próbie uzyskania lepszych wyników z użyciem dostarczonej w materiałach implementacji SA (a właściwie DA – Dual Annealing) w Python, która posiadała większą liczbę różnorodnych parametrów od wersji C++. Testowanie nie było w tym przypadku tak dokładne i zostało wykonane ręcznie z jednoczesną obserwacją wyników i zachowania. Dobór parametrów był oparty o losowy przegląd, a później również o podstawowa analizę dokumentacji użytej funkcji dual_annealing z pakietu scipy.

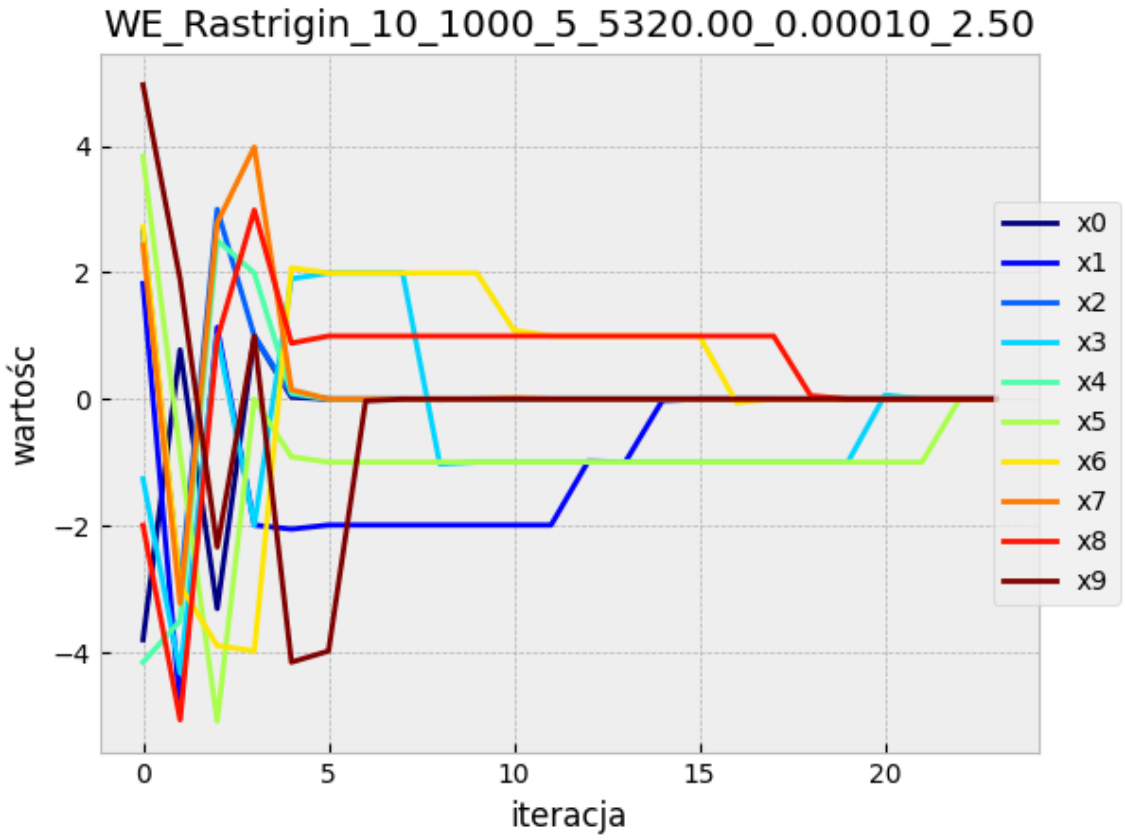
Test został wykonany na 10 – wymiarowej funkcji Rastragina, ponieważ implementacja C++ poradziła sobie z nią najgorzej.

Po paru eksperymentach z parametrami udało się uzyskać najlepszy wynik **y = 2.842170943040401e-14 (24 iteracje)** dla parametrów x podobnie bliskich zeru. Uzyskiwane wyniki miały podobny rząd wielkości przy wielu uruchomieniach z różnych jądrem generatora losowego.

Te wyniki zostały uzyskane dla ustawień:

- L.iteracji = 1000
- Tempteratura start = 5320
- Współczynnik restartu procesu (reannealing) rest_ratio = restart_temp_ratio = 0.0001
- Skok w przestrzeni rozwiązań Vis = 2.5
- Rozkład prawdopod. Akceptacji Acc = -5
- Warunek stopu przeszukiwania lokalnego Max_tries = 1e7

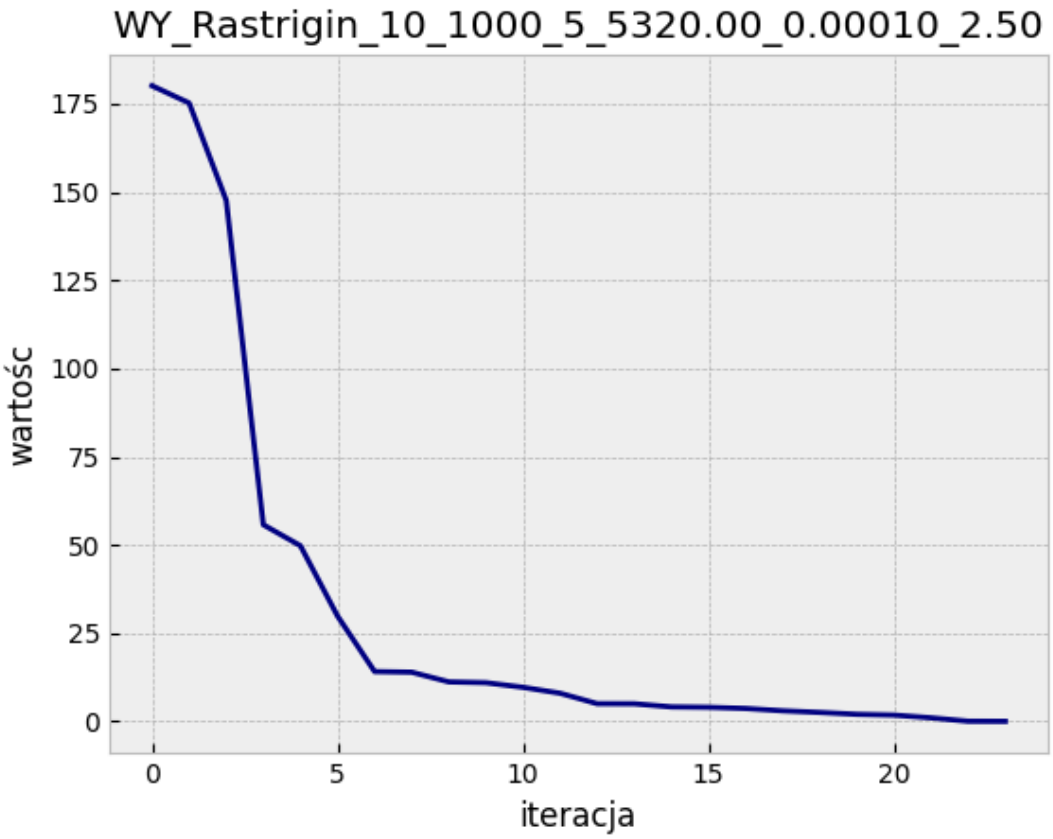
Dało się uzyskać wyniki podobnej jakości w krótszym czasie dla większego rest_ratio = 0.01, jednak wtedy raz na jakiś czas dochodziło do osiągnięcia słabego wyniku w okolicach 1. Wyższa wartość powodowała częstsze restartowanie procesu, co sprawia że przy niefortunnych czynnikach losowych częstsze restarty mogły powodować brak zbieżności.



Wykres przedstawiający wartości parametrów x w zależności od iteracji

Na wykresie jest zauważalny ogólnie dobry zbieg każdego parametru do 0 (w okolice 0), które w tej funkcji jest optymalne. W tytule zawarte są te same parametry co w umieszczonej wyżej liście oraz seed_num, które mimo ustawienia po prostu nie było używane. Widać jednak wpływ losowej akceptacji innych rozwiązań – szczególnie na początku pracy algorytmu zauważalne są sytuacje, gdzie

wartość parametrów ulega tymczasowemu pogorszeniu (np. x7 w okolicach 4 iteracji jest dalsze od 0 niż dla 1, czy 2 iteracji).



Zależność wyniku (y = wartość) od iteracji

W przeciwieństwie do wersji C++ w tym przypadku zauważalna jest bardzo dobra zbieżność. Wartość wyniku Y szybko spada we wczesnych iteracjach – wykres jest bardzo stromy. Idąc dalej, zmiany są co raz mniejsze.

Co do samej liczby iteracji to była ona znacznie mniejsza, lecz wynika to z faktu że są one inaczej liczone. W użytym Dual Annealing proces poszukiwań jest połączeniem globalnej eksploracji oraz lokalnej eksploatacji. Prawdopodobnie jedna iteracja algorytmu jest więc połączeniem dwóch oddzielnych poszukiwań. Sam czas trwania wynosił ok. 2 – 3 sekundy, co wciąż jest szybkie dla użytkownika, jednak wolniejsze niż wersja C++ (pomijając eksperymenty z ogromną liczbą iteracji).

3.1 Wnioski

Można wysnuć następujące wnioski:

- Bardziej zaawansowany algorytm Dual Annealing jest skuteczniejszy od klasycznego SA, posiada więcej parametrów, ale jest trochę wolniejszy
- DA łączy eksploracje z eksploatacją przez co nie jest tak podatny na minima lokalne oraz przedwczesne spowolnienie jak banalny SA
- Najbardziej znaczącymi parametrami DA okazały się być Temperatura startowa initial_temp oraz restart_temp_ratio

- Zbieżność wartości parametrów x oraz y jest znacznie lepsza dla DA niż SA. W DA zauważalny jest stabilny postęp wraz z iteracjami (ew. czasami tymczasowe pogorszenie) zamiast losowych małych popraw jak dla SA
- Wydaje się, że największą wadą testowanej implementacji SA jest brak możliwości dostosowania zmian temperatury – jest możliwe tylko domyślne ustawienie, gdzie temperatura jest zmniejszana o wartość Tstart/n_iter. Ze względu na podobieństwo w działaniu można ocenić, że jest podobnie trywialne, a zarazem problematyczne co stałokrokowe metody gradientowe.
- W DA połączenie eksploracji z eksploatacją oraz dokładniejsza parametryzacja pozwala na bardziej konsekwentne osiąganie porządnych wyników, niż trywialne SA, które można nazwać „ulepszoną zgadywanką” ze względu na wpływ losowości na jego wyniki