AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PRACA DYPLOMOWA INŻYNIERSKA

pt.

„Implementacja procedur optymalizacji przystosowanych   
do obliczeń w środowiskach masowo równoległych”

Imię i nazwisko dyplomanta: **Jakub Karamański**

Kierunek studiów: **Informatyka Stosowana**

Nr albumu: **268958**

Promotor: dr hab. inż. Łukasz Rauch

Recenzent: dr inż. Piotr Kustra

Podpis dyplomanta: Podpis promotora:

Kraków 2017

„Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia   
4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90,  
poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do  
autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega  
grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej  
karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy  
utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo  
publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub  
nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211  
ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r.  
poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny  
uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed  
komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym  
dalej "sądem koleżeńskim"”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż  
wymienione w pracy.”

Kraków, dnia ……………… Podpis dyplomanta………………….

**Spis treści**

[1. Wstęp 1](#_Toc471631460)

[2. Opis wybranych algorytmów 2](#_Toc471631461)

[2.1. Metoda Hook’a-Jeevesa 2](#_Toc471631462)

[2.2. Metoda Nelder’a-Mead’a 3](#_Toc471631463)

[3. Idea zrównoleglania algorytmów 7](#_Toc471631464)

[3.1. Przegląd metod 7](#_Toc471631465)

[3.2. Zrównoleglenie metody Hook’a-Jeeves’a 8](#_Toc471631466)

[3.3. Zrównoleglenie metody Nelder’a-Mead’a 9](#_Toc471631467)

[4. Projekt oprogramowania 11](#_Toc471631468)

[4.1. Diagram klas 11](#_Toc471631469)

[4.2. Szczegóły implementacji 14](#_Toc471631470)

[4.2.1. Implementacja metody Hook’a–Jeeves’a 14](#_Toc471631471)

[4.2.2. Implementacja metody Nelder’a–Mead’a 16](#_Toc471631472)

[5. Wyniki 18](#_Toc471631473)

[5.1. Specyfikacja sprzętu 18](#_Toc471631474)

[5.2. Funkcje Testowe 18](#_Toc471631475)

[5.2.1. Funkcja testowa Booth’s 18](#_Toc471631476)

[5.2.2. Funkcja testowa Baley’s 19](#_Toc471631477)

[5.2.3. Funkcja testowa Ackley’s 20](#_Toc471631478)

[5.2.4. Funkcja testowa Rastrigin 2D 21](#_Toc471631479)

[5.2.5. Funkcja testowa Rastrigin 5D 23](#_Toc471631480)

[5.3. Opracowanie wyników 24](#_Toc471631481)

[6. Podsumowanie 26](#_Toc471631482)

[7. Literatura 27](#_Toc471631483)

# Wstęp

Wraz z wzrostem techniki pojawiło się zapotrzebowanie na środowiska,  
które oferowały bardzo dużą moc obliczeniową, którą można wykorzystać przy analizie danych czy obliczeniu skomplikowanych równań. Środowiska masowo równoległe są tworzone   
i wykorzystywane m.in. do obliczeń wielkoskalowych, symulacji numerycznych za pomocą złożonych modeli nieliniowych czy też wykonywania obliczeń wielo-iteracyjnych.

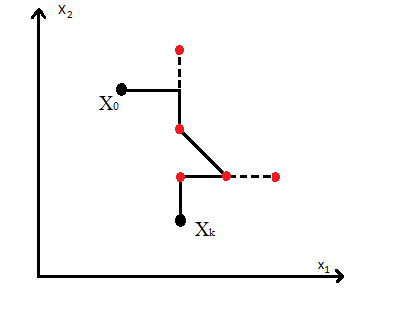
W tej pracy podjęto problem adaptacji procedur optymalizacji, które w swojej naturze nie są równoległe, czyli metod, które w pojedynczym kroku wykonują obliczenia tylko dla jednego punktu. Przystosowanie algorytmów do pracy z serwerami obliczeniowymi przyniosłoby znaczny wzrost wydajności. Algorytmy takie jak genetyczny czy rój cząstek są w naturze równoległe więc łatwo przystosować je do obliczeń na maszynach wielordzeniowych lecz w tej pracy zostały przedstawione takie, które są trudne do zrównoleglenia ale przy zastosowaniu odpowiednich modyfikacji mogą okazać się bardzo skuteczne, co zostało również zbadane w tej pracy.

W drugim rozdziale zostały opisane wybrane algorytmy, które zostały zrównoleglone. Trzeci rozdział przedstawia ideę zrównoleglania algorytmów. Zostały w nim zawarte przykładowe zrównoleglania przedstawionych algorytmów. W rozdziale czwartym znalazł się projekt oprogramowania oraz szczegóły implementacji. Piąty rozdział zawiera analizę zmodyfikowanych algorytmów z wykorzystaniem kilku standardowych problemów (funkcji) testowych. W tym rozdziale zostało, także zawarte opracowanie wyników, w którym porównano czasy działania poszczególnych algorytmów dla funkcji testowych.   
Podsumowanie pracy zostało przedstawione w rozdziale szóstym, w którym zebrano najważniejsze wnioski.

# Opis wybranych algorytmów

## Metoda Hook’a-Jeevesa

Algorytm Hook’a-Jeeves’a zalicza się do iteracyjnych metod optymalizacji. Składa się z sekwencji skoków z punktu bazowego – krok próbny, a następnie z ruchów – krok roboczy, które zapewniają następny punkt bazowy do zbadania. W kroku próbnym wykonuje lokalne wyszukiwanie w jednym kierunku poprzez zmianę parametru lub parametrów punktu bazowego. Jeśli wartość funkcji w tym punkcie jest lepsza niż wartość poprzednia to algorytm wybiera ten nowy punkt jako punkt bazowy. W przeciwnym razie algorytm wykonuje wyszukiwanie w przeciwnym kierunku i jeśli wynik jest lepszy od poprzedniego to nowy punkt zostaje punktem bazowym. Algorytm cały czas przechowuje wartość początkową i sprawdza czy wartość jest lepsza czy gorsza. Gdy wszystkie parametry zostały zbadane algorytm przechodzi do kroku roboczego. W kroku roboczym każdy z parametrów zostaje zwiększony   
o stałą. Celem jest przesunięcie punktu bazowego w kierunku, który poprawiłby wynik z kroku próbnego. Następnie funkcja zostaje oceniana w nowym miejscu. Jeśli wartość jest lepsza to nowy punkt staje się punktem bazowym w przeciwnym razie przemieszczenie jest ignorowane, a stała zostaje zredukowana, żeby zmniejszyć przemieszczenie układu [1]. Kroki te są powtarzane aż do momentu, kiedy nie zostanie spełniony warunek stopu.



**Rys. 2.** Graficzne przedstawienie działania metody Hook’a – Jeevesa

Źródło : opracowanie własne

Na rysunku 2 można zobaczyć graficzne przedstawienie działania metody Hook’a-Jeevesa. Punkt początkowy znajduje się w *xo*, z którego czarną linią zostały zaznaczone kroki pomyślne czyli takie, które dały mniejszą wartość funkcji, a linie przerywane wraz   
z zaznaczonymi na czerwono punktami to kroki próbne, gdzie wartość funkcji w tych punktach była większa, te kroki uznajemy za niepomyślne. Kroki próbne sprawdzane są w dwóch kierunkach: góra i dół, aby uznać, który będzie najlepszy. Etap roboczy wykonywany jest dopiero wtedy, gdy przynajmniej jeden z kroków etapu próbnego zakończył się sukcesem, czyli osiągnięcia wartości funkcji mniejszej niż w punkcie bazowym. Skoki próbne oraz skoki robocze zawsze wykonywane są, aż do momentu znalezienia ekstremum funkcji lub   
w przypadku jej nie znalezienia do określonego warunku stopu.

## Metoda Nelder’a-Mead’a

Algorytm simpleksów Neldera-Meada (ang. Downhill Simplex, Amoeba), jest metodą wykorzystywaną do wyznaczenia ekstremum funkcji . Główną zaletą algorytmu Neldera-Meada jest bardzo niska złożoność obliczeniowa w przypadku niewielu rozbudowanych funkcji celu. Algorytm polega na utworzeniu w przestrzeni En+1 *n*-wymiarowego simpleksu o *n*+1 wierzchołkach. Wykonuje się to po to, aby można było go wpisać w powierzchnię reprezentująca badaną funkcję celu. Pierwszym krokiem, jaki należy wykonać, to wyliczanie punktów wierzchołkowych simpleksu *Qj*, przy czym zakładana jest pewną odległość pomiędzy wierzchołkami – tak zwany krok. W kolejnych iteracjach dokonuje się wymiany „najgorszego” wierzchołka i zostaje budowany nowy simpleks, aż odległość pomiędzy jego wierzchołkami   
w pobliżu poszukiwanego ekstremum funkcji będzie mniejsza od założonej dokładności obliczeń. Kryterium zbieżności to dokładność obliczeń w tej metodzie [2]. Podczas kolejnych iteracji wykonujemy operacje, które mają na celu osiągnięcie najniższej wartości funkcji celu:

1. Odbicie punktu *Qh*względem *Q’*,

Początkowo należy wybrać najgorszy punkt (punkt, w którym funkcja ma największą wartość) i odbić go względem punktu *Q’*, który wyznaczamy z wzoru:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Następnie zostaje obliczona odległość pomiędzy punktami *Q’*, a punktem *Qh*:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

Ostatnim etapem jest obliczenie punktu *Q\**

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

gdzie: *α* – współczynnik odbicia (*α* > 0)

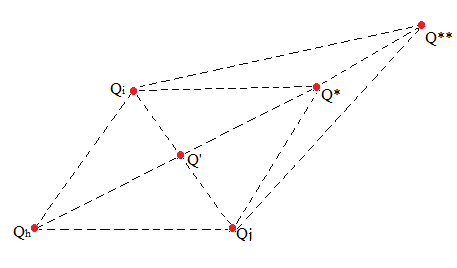
|  |  |
| --- | --- |
|  |  |
| **Rys. 3.** Graficzne przedstawienie odbicia punktu  gdzie współczynniki odbicia α = 1  Źródło : opracowanie własne | **Rys. 4.** Graficzne przedstawienie odbicia punktu  gdzie współczynnik odbicia α = 0.5  Źródło : opracowanie własne |

Na rysunku 3 i 4 zostało graficznie pokazane odbicie punktu *Qh* względem punktu *Q’*.

1. Ekspansja punktu *Q\*\** względem *Q’*,

Jeśli poprzedni krok został zakończony pomyślnie czyli wartość funkcji w punkcie *Q\** jest najmniejsza wykonujemy ekspansję czyli przejście o zadaną odległość   
w tym samym kierunku co w poprzednim kroku:

|  |  |  |
| --- | --- | --- |
|  |  | (4) |



**Rys. 5.** Graficzne przedstawienie ekspansji punktu

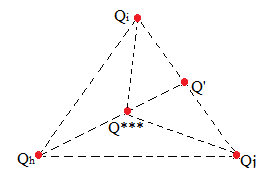
Źródło : opracowanie własne

Gdzie:  
*γ* – współczynnik ekspansji (*γ* > 0)

1. Kontrakcja punktu *Qh* względem *Q’*,

Jeśli odbicie nie zostało wykonane wykonujemy kontrakcję – nowy punkt otrzymamy wykonując odbicie do środka simpleksu względem punktu *Q’*.

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

****

**Rys. 6.** Graficzne przedstawienie kontrakcji punktu

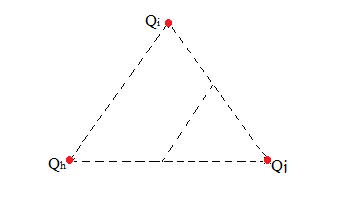
Źródło : opracowanie własne

Gdzie:  
*β* – współczynnik kontrakcji (0 < *β* < 1)

1. Redukcja simpleksu,

Redukcja simpleksu została wprowadzona po to aby zapobiec upadkom algorytmu, obliczaniu złych wartości oraz poprawienie dokładności obliczeń.

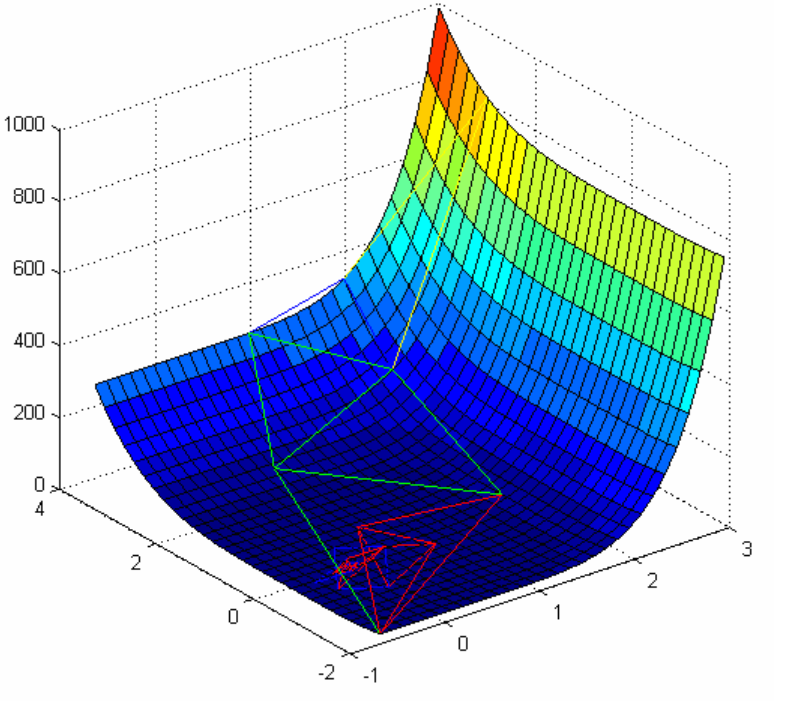
|  |  |  |
| --- | --- | --- |
|  |  | (6)  (7) |



**Rys. 7.** Graficzne przedstawienie redukcji simplexu

Źródło : opracowanie własne

W algorytmie Neldera-Meada dobiera się wartości współczynników w następujący sposób:   
*α* = 1, *β* = 0.5, *γ* = 2. Na rysunku 8 można zobaczyć obraz zminimalizowanej funkcji 2D:



**Rys. 8.** Graficzne przedstawienie działania metody Nelder’a-Mead’a  
dla funkcji dwuwymiarowej

Źródło : [3]

Podsumowując w metodzie bezgradientowej, jaką jest algorytm Neldera-Meada   
na dokładność rezultatów oraz szybkość wpływają:

1. skomplikowanie funkcji celu,
2. tolerancja algorytmu, która gwarantuje zakończenie obliczeń.

Więcej informacji na temat tego algorytmu znajduje się w [4].

# Idea zrównoleglania algorytmów

Algorytmy wykonujące tylko jedne obliczenia w jednej iteracji, które później wykorzystujemy do rozwiązywania skomplikowanych równań matematycznych lub opisu zjawisk fizycznych nie rozwiążą tych równań w krótkim czasie. Wykonywanie dwóch obliczeń jednocześnie teoretycznie powinno skrócić czas o połowę, a dodając kolejne jednostki obliczeniowe jesteśmy w stanie jeszcze skrócić ten czas [5]. Algorytmy, które zostały opisane w rozdziale wyżej nie są typowymi algorytmami, które łatwo można zrównoleglić.   
Ich podstawowa wersja w jednej iteracji liczy tylko wartość dla jednego punktu. Aby zrównoleglić takie algorytmy trzeba było zastanowić się, który moment będzie najlepszy do zrównoleglenia i przyniesie nam najlepsze wyniki. Zrównoleglenie tych algorytmów miało na celu znacząco przyspieszyć czas obliczeń, co było najważniejszym punktem do osiągnięcia.

## Przegląd metod

Zrównoleglenie algorytmów w dzisiejszych czasach jest ważnym elementem programowania. W literaturze czy intrenecie można znaleźć wiele przykładów implementacji równoległej zarówno algorytmu Hook’a-Jeeves’a oraz Nelder’a-Mead’a. Przykładowe metody zrównoleglenia algorytmu Hook’a-Jeeves’a:

1. jednoczesne wykonywanie skoku próbnego w kilku kierunkach i powtarzanie tej operacji,
2. wykonywanie skoku próbnego z różną długością skoku na osobnych wątkach,
3. przemieszczanie się w jednym kierunku i równolegle sprawdzać prawdopodobieństwo czy następny krok przyniesie mniejszą wartość.

Metody zrównoleglenia algorytmu Nelder’a-Mead’a:

1. wykonywanie operacji odbicia, ekspansji i kontrakcji z różnymi wartościami współczynników jednocześnie,
2. osobne przypisanie każdemu z wątków operacji odbicia, ekspansji i kontrakcji, po zakończeniu czynności porównać ze sobą wartości funkcji celu i wybrać najmniejszą.

Jest to tylko kilka przykładów zrównoleglenia tych metod, każda z nich ma na celu zmniejszenie czasu obliczeń oraz podać dokładniejszą wartość funkcji celu.

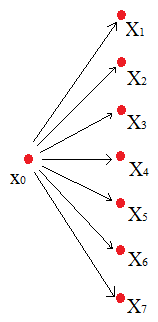
## Zrównoleglenie metody Hook’a-Jeeves’a

W niniejszej pracy algorytm zrównoleglono innymi sposobami niż przedstawionymi w rozdziale powyżej, a mianowicie:

1. tworzyć losowe ścieżki (w różnych kierunkach) w zależności od wartości losowej zmiennej zmieniającej się przy każdej iteracji (dodając do obliczeń kilka punktów w jednym kierunku), przeliczyć wszystkie punkty naraz i wybrać najlepszy,
2. tworzyć skoki w jednym kierunku z pewnymi zadanymi odchyleniami   
   i sprawdzić, w którym punkcie wartość funkcji jest najmniejsza.

Obliczenia, jakie zostały wykonane potwierdziły słuszność wybranej drogi i przyniosły bardzo dobre efekty. Ilość losowych ścieżek z wariantu punktu a) była uzależniona od ilości wymiarów. Następnie punkty te zostały przeliczone na osobnych wątkach i porównane, co pozwoliło przyspieszyć czas obliczeń w porównaniu do implementacji tego algorytmu bez modyfikacji. Dokonano zrównoleglenia zarówno kroku próbnego jak i kroku roboczego. Negatywny wpływ na działanie tego wariantu implementacji mógł mieć czynnik losowy, ponieważ nigdy tak naprawdę nie wiadomo, w którym kierunku algorytm osiągnie lepszą wartość i kiedy będzie można zakończyć obliczenia.

Wariant z punktu b), gdzie tworzyliśmy dużo większą ilość punktów, pozwolił na jeszcze większe skrócenie czasów obliczeń.



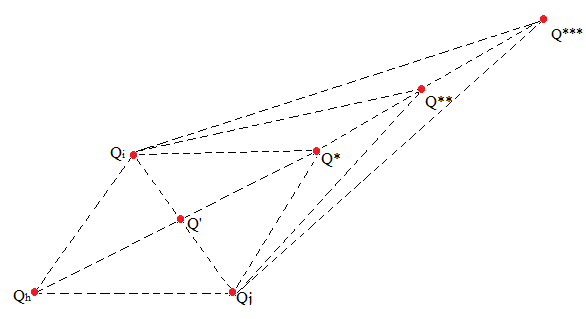
**Rys. 9.** Graficzne przedstawienie zrównoleglenia metody Hook’a Jeeves’a  
skoki z zadanym odchyleniem

Źródło : opracowanie własne

Na rysunku 9 można zobaczyć działanie zrównoleglenia algorytmu Hook’a Jeevesa.   
Łatwo zauważyć jak wygląda odchylenie od wybranego przez nas punktu początkowego.   
Po przeliczeniu wszystkich punktów algorytm porówna wartości funkcji celu i wybierze najmniejszą wartość. Dodanie do przeliczenia tylu punktów, ile jest dostępnych wątków   
w maszynie równoległej, pozwoli w najlepszy sposób wykorzystać całą moc obliczeniową. Dodanie zbyt dużej ilości punktów już nie będzie przynosiło znacznej poprawy w wydajności, a niewykorzystanie pełnej mocy obliczeniowej wydłuży czas obliczeń danego problemu.

## Zrównoleglenie metody Nelder’a-Mead’a

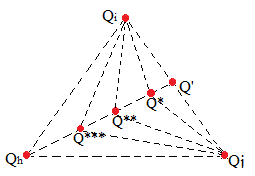
Algorytm Nelder'a - Mead'a został zrównoleglony poprzez jednoczesne przeliczenie wielu punktów, które zostały uzyskane poprzez wielokrotne wykonanie odbicia, ekspansji oraz kontrakcji w jednej iteracji. Wykorzystanie takiego rodzaju zrównoleglenia pozwoliło wykonać wiele obliczeń wartości funkcji w punktach, dzięki czemu czas obliczeń znacznie się zmniejszył. Ilość punktów powstałych przez wielokrotne odbicie, ekspansje oraz kontrakcję podaje użytkownik. W podstawowej wersji tego algorytmu funkcja celu jest liczona osobno dla każdego punktu powstałego np. po odbiciu.



**Rys. 10.** Graficzne przedstawienie wielokrotnego odbicia w algorytmie Nelder’a – Mead’a

Źródło : opracowanie własne

Na rysunku 10 pokazano odbicie poprzez wykonanie trzech skoków w tym samym kierunku. Następnie funkcja oblicza wartość funkcji celu dla wszystkich punktów oraz zwraca ten punkt gdzie jej wartość jest najmniejsza. Jeśli wielokrotne odbicie spowodowało mniejszą wartość funkcji celu wykonujemy wielokrotną ekspansję, czyli wielokrotnie przenosimy się w tym samym kierunku, obliczamy wszystkie punkty i sprawdzamy czy, któryś z nich zwróci minimum. W tym przypadku wielokrotna ekspansja może być także zobrazowana jako rysunek 10. Wybieramy punkt i przenosimy się kilkukrotnie w jednym kierunku.



**Rys. 11.** Graficzne przedstawienie wielokrotnej kontrakcji w algorytmie Nelder’a – Mead’a

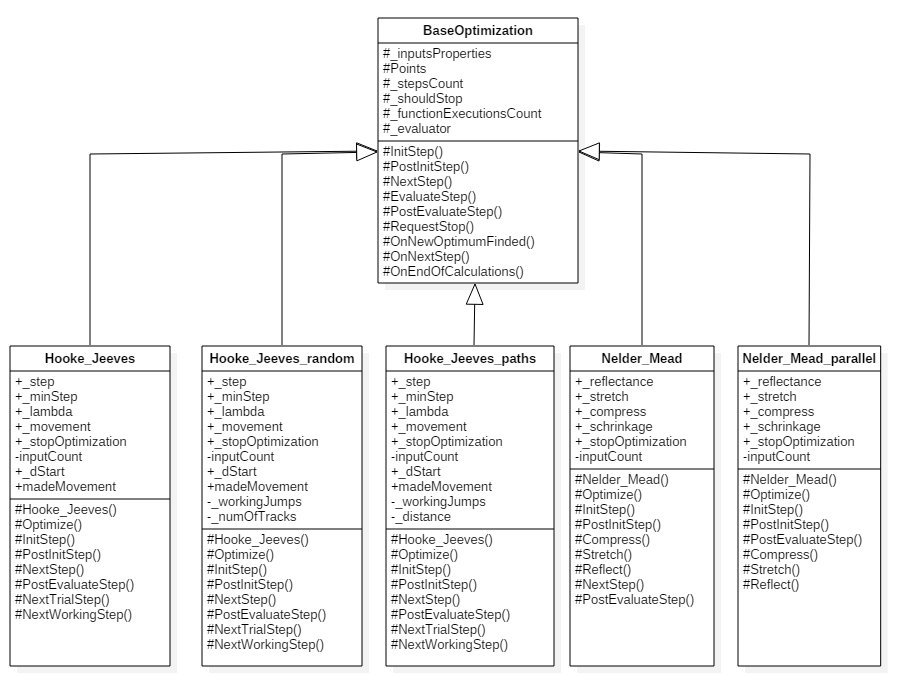
Źródło : opracowanie własne

W razie niepowodzenia – wartość funkcji celu po kilkukrotnej ekspansji jest większa – wykonujemy wielokrotną kontrakcję, którą przedstawia rysunek 11.

# Projekt oprogramowania

## Diagram klas

Wszystkie implementacje algorytmów zostały napisane w osobnych klasach dla czytelności kodu.

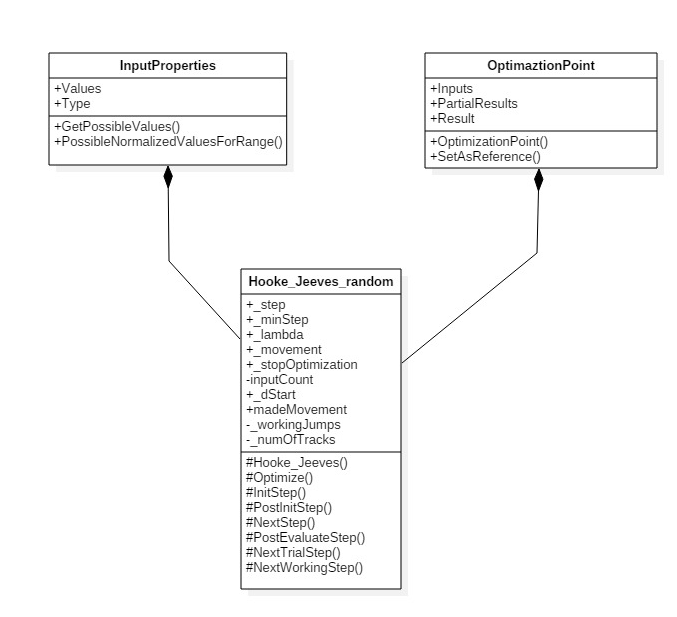


**Rys. 12.** Diagram klas

Źródło : opracowanie własne

Każda z klas dziedziczy po klasie abstrakcyjnej jaką jest BaseOptimization. Ta klasa jest odpowiedzialna za kolejne kroki algorytmu oraz przeliczanie punktów. Dzięki temu klasy pochodne nadpisują metody wirtualne tj. InitStep, PostInitStep, NextStep, EvaluateStep oraz PostEvaluateStep. W implementacji algorytmów zostały zawarte własne atrybuty jak i funkcje, które są potrzebne do wykonania algorytmu.

Do reprezentowania punktów w przestrzeni oraz wartości początkowych została wykorzystana klasa OptimizationPoint oraz InputProperties. Na przykładzie metody Hook’a-Jeeves’a w wersji równoległej został przedstawiony poniższy diagram:

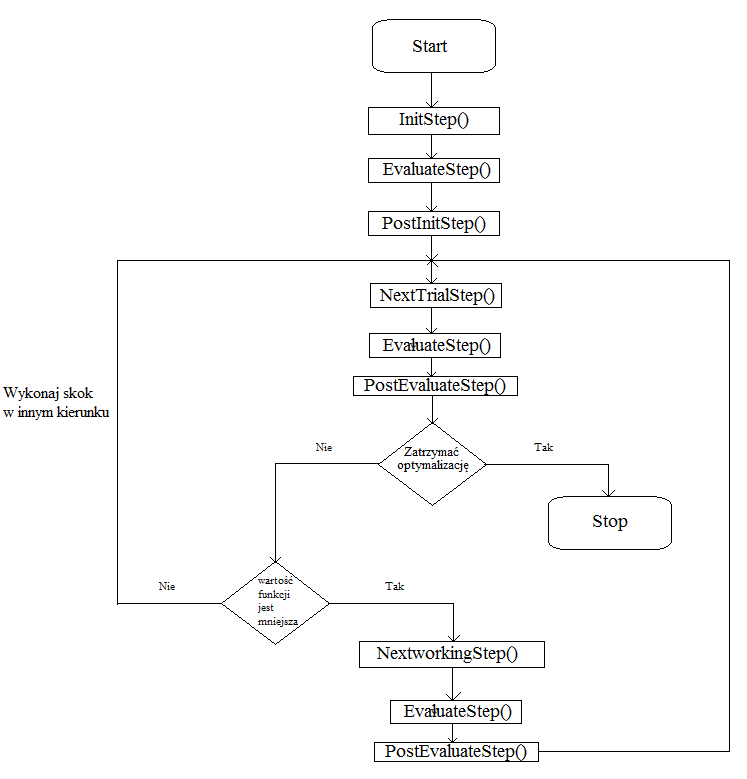


**Rys. 13.** Diagram klas

Źródło : opracowanie własne

Na rysunku 13 przedstawiona została klasa Hook\_Jeeves\_random wykorzystuje OptimizationPoint do przedstawienia parametrów danego punktu. Dzięki tej klasie możemy wykonywać działania na punktach w przestrzeni.

Poniżej przestawiony został schemat blokowy sekwencji etapów dla algorytmu Hook’a-Jeeves’a. Każdy z algorytmów został zaimplementowany według schematu przedstawionego na rysunku 12.

****

**Rys. 12.** Schemat blokowy metody Hook’a-Jeeves’a

Źródło : opracowanie własne

Funkcja InitStep() inicjuje punkty początkowe które zostają później przeliczone w metodzie EvaluateStep i porównane w PostInitStep. Następnie w pętli zostają wykonywane skoki próbne oraz kroki robocze wraz z sprawdzeniem czy obliczenia nie powinny zostać przerwane   
z powodu warunku stopu. Funkcja EvaluateStep() to funkcja, która włącza funkcję Evaluator:

*Listing nr. 1:*

|  |
| --- |
| Private static void Evaluator(List<Optimization.Core.OptimizationPoint> points)  {  Paralel.For(0, points.Count, (i) =>  {  points[i].PartialResults = rastrigin(points[i].Inputs);  });  } |

Funkcja Evaluator przedstawiona na Listingu nr. 1 przyjmuje jako parametr listę punktów,  
a następnie równolegle w pętli punkty zostają obliczone dla wybranej funkcji.

## Szczegóły implementacji

Implementacja obu algorytmów została wykonana w języku programowania C#. Język ten jest prosty, nowoczesny oraz zorientowany obiektowo. C# zapewnia przyjazny użytkownikowi interfejs, szybkie wykonywanie aplikacji, wysokie bezpieczeństwo oraz może działać wszędzie tam gdzie jest środowisko .NET bez instalacji pełnego systemu [6]. Badane funkcje są bardzo kosztowne obliczeniowo, że algorytmy zaimplementowane w języku C++ nie przyniosłyby realnego zysku natomiast zaletą C# jest szybka implementacja oprogramowania. Algorytm Hook’a – Jeeves’a oraz Nelder’a Mead’a zostały zrównoleglone tak, aby w zależności od złożoności funkcji oraz ilości punktów do przeliczenia,  
metody mogły zostać wykonane na tylu wątkach ile jest ich dostępnych w maszynie równoległej Do reprezentacji punktów w przestrzeni została wykorzystana klasa OptimizationPoint, która pozwala na przechowywanie punktów w liście, utrzymanie współrzędnych oraz rezultatów obliczeń funkcji celu dla danego punktu, co ułatwia późniejsze porównywanie oraz odnajdywanie najlepszego punktu.

## Implementacja metody Hook’a–Jeeves’a

W metodzie Hook’a – Jeeves’a w wersji nierównoległej krok próbny został zaimplementowany w następujący sposób:

*Listing nr. 2:*

|  |
| --- |
| newFx.Inputs[i] -= \_step[i]; // przenosimy się w lewo  Points.Add(newFx); // dodajemy punkt do przeliczenia  EvaluateStep(); // Przeliczenie punktu |

Na listingu 1 można łatwo zauważyć, że w tym przypadku z każdym nowym dodanym punktem przeliczamy go osobno, a następnie wykonujemy operację porównania, aby znaleźć wartość najmniejszą. Krok roboczy został przedstawiony na listingu nr. 3.

*Listing nr. 3:*

|  |
| --- |
| newFx = new Core.OptimizationPoint(G\_Best); for (int i = 0; i < inputCount; i++)  newFx.Inputs[i] += \_lambda \* \_dStart[i]; Points.Add(newFx); //dodajemy punkt do przeliczenia EvaluateStep(); //liczymy punkt |

Krok roboczy pokazany na listingu nr. 2 w wersji podstawowej wykonuje tylko jeden skok i liczy wartość funkcji celu tylko w tym punkcie.

Równoległa wersja tego algorytmu zakłada tworzenie losowych ścieżek ze skoków próbnych oraz skoków roboczych. Następnie wszystkie punkty uzyskane ze ruchów przeliczamy jednocześnie i wybieramy najlepszy punkt gdzie wartość funkcji będzie najmniejsza. Przykład skoku próbnego przedstawiony jest na listingu nr. 4:

*Listing nr. 4:*

|  |
| --- |
| int direction = Helpers.NextInt(0, 1); // funkcja losująca kierunek if (direction == 0){  newFx.Inputs[i] -= \_step[i]; // przenosimy się w lewo   Points.Add(new Core.OptimizationPoint(newFx)); // dodajemy punkt  }  else if (direction == 1){  newFx.Inputs[i] += \_step[i]; // przenosimy się w prawo  Points.Add(new Core.OptimizationPoint(newFx)); // dodajemy punkt  } |

W listingu nr. 4 można zauważyć, że zostaje wylosowany kierunek, w którą stronę będziemy wykonywać skok. Skoki mogą zostać wykonane naprzemiennie, a ilość skoków zależna jest od ilości zmiennych funkcji. Kolejna czynność, którą należy wykonać to dodać punkty do przeliczenia. Odpowiednia funkcja która przyjmuje listę punktów zwróci nam wartość funkcji celu w najlepszym punkcie oraz współrzędne punktu. Funkcja obliczająca jest przystosowana do pracy na wielu wątkach i każdy punkt jest liczony osobno.

Kolejna z równoległych wersji skoku próbnego:

*Listing nr. 5:*

|  |
| --- |
| // generujemy wiele ruchów w lewo  newFx = new Core.OptimizationPoint(G\_Best); // kopia chwilowa  SinglePointJump(newFx, i, -1 \* l); // skok w lewo z odchyleniem  Points.Add(newFx); // dodanie punku |

W wersji zawartej w listingu 5 skoki próbne wykonywane są z podanym przez użytkownika odchyleniem. Funkcja SinglePointJump jako parametry, przyjmuje punkt bazowy, kierunek oraz długość skoku. Na listingu nr. 6 widoczna jest funkcja SinglePointJump:

*Listing nr. 6:*

|  |
| --- |
| void SinlePointJump(Core.OptimizationPoint point, int input, double dist\_multiplier){  for ( int k = 0; k < this.\_inputCount; ++k){  //przenosimy się w lewo lub prawo  if( k == input)  newFx.Inputs[k] += \_step[k] \* dist\_multipler;  else{  //delikatne losowe odchylenie w innych kierunkach  newFx.Inputs[k] += Helpers.NextDouble(-0.1, 0.1) \* \_step[k] \* dist\_multiplier  }  } } |

Funkcja SinglePointJump zwraca punkty odchylone od punktu bazowego tak jak zostało to przedstawione na rysunku 9. W zależności od ilości parametrów wejściowych tyle odchyleń zostanie wykonanych i tyle punktów zostanie dodanych do przeliczenia każdemu przypisując osobny wątek.

## Implementacja metody Nelder’a–Mead’a

W metodzie Nelder’a – Mead’a wykonywane są cztery operacje w każdej iteracji: odbicie, ekspansja, kontrakcja oraz kurczenie simpleksu. Na poniższym listingu zostało przedstawione odbicie niezrównoleglone:

*Listing nr 7:*

|  |
| --- |
| private OptimizationPoint Reflect(OptimizationPoint centerPoint, OptimizationPoint point)  {  return centerPoint + (centerPoint - point) \* \_reflectance;  } |

Funkcja Reflect przyjmuje jako argumenty funkcji środek ciężkości simpleksu oraz punkt odbijany – w tym przypadku najgorszy punkt. Zwraca nam punkt uzyskany po odbiciu.  
Aby można było wykonywać operacje na współrzędnych wybranych punktów zostały zaimplementowane operatory sumy, różnicy, iloczynu i ilorazu.

*Listing nr 8:*

|  |
| --- |
| public static OptimizationPoint operator + (OptimizationPoint p1, OptimizationPoint p2)  {  return new OptimizationPoint(p1.Inputs.Select((x, i) => x + p2.Inputs[i]).ToList());  } |

Na listingu 8 został pokazany przeładowany operator dodawania dla współrzędnych wybranych punktów.

W wersji równoległej odbicie wygląda następująco:

*Listing nr 9:*

|  |
| --- |
| public List<OptimizationPoint> Reflect(OptimizationPoint centerPoint, OptimizationPoint point, int count) {  List<OptimizationPoint> points = new List<OptimizationPoint>();  for (int i = 0; i < count; ++i) {  points.Add(centerPoint + (centerPoint - point) \* \_reflectance \* (i + 1.5));  }  return points;  } |

Metoda przestawiona na listingu nr. 9 przyjmuje środek ciężkości, punkt odbijany oraz ilość wykonanych odbić. Funkcja zwraca nam listę punktów wykonaną przez wielokrotne odbicie, które następnie trafiają do funkcji obliczającej, która zwróci punkt z najmniejszą wartością funkcji celu oraz jego współrzędnymi.

Listing nr. 10 prezentuje ekspansję wykonaną wielowątkowo:

*Listing nr 10:*

|  |
| --- |
| public List<OptimizationPoint> Stretch(OptimizationPoint centerPoint, OptimizationPoint point, int count) {  List<OptimizationPoint> points = new List<OptimizationPoint>();  for (int i = 0; i < count; ++i) {  points.Add(centerPoint + (centerPoint - point) \* \_stretch \* (i + 1.5));  }  return points;  } |

Ekspansja przedstawiona na listingu nr. 10 została zaimplementowana w oparciu o metodę odbicia. W taki sam sposób została napisana czynność wykonująca kontrakcję.  
Każda z operacji różni się w niewielkim stopniu wzorem oraz współczynnikiem.

# Wyniki

## Specyfikacja sprzętu

. Każdy z algorytmów został przetestowany na procesorze Intel(R) Core(TM) i5-4210 U CPU @ 1.70 GHZ,a w trybie Hyper Threading dysponuje 2.6 GHz. Procesor jest oparty o 64-bitową architekturę i posiada 3MB pamięci cache. Należy do rodziny procesorów niskonapięciowych tzw. ultramobilnych wykonany w procesie technologicznym 22 nm. Charakteryzuje się dwoma rdzeniami, czterema wątkami oraz dysponuje 8 GB pamięci RAM o taktowaniu 1600 MHz.

## Funkcje Testowe

Algorytmy przedstawione w tej pracy zostały poddane badaniom przy użyciu funkcji testowych, które oceniają cechy algorytmów optymalizacji. Każdy z algorytmów z powodu bardzo małego czasu wykonania został przetestowany stu krotnie co pozwoliło opracować dokładniejsze wyniki. Kryteriami zatrzymania optymalizacji dla każdych z algorytmów dla funkcji testowych były:

1. maksymalny dopuszczalny błąd e = 0,01,
2. maksymalna liczba iteracji wynosząca 100.

## Funkcja testowa Booth’s

Pierwsza z testowych funkcji ma jedno minimum dla *f* (1, 3) = 0. Rysunek 17 pokazuje wykres funkcji:

|  |
| --- |
| File:Booth's function.pdf  **Rys. 15.** Funkcja testowa : Beale's Źródło : [7] |

Wzór funkcji:

|  |  |
| --- | --- |
|  | (9) |

W tabeli 1 zostały przedstawione wyniki:

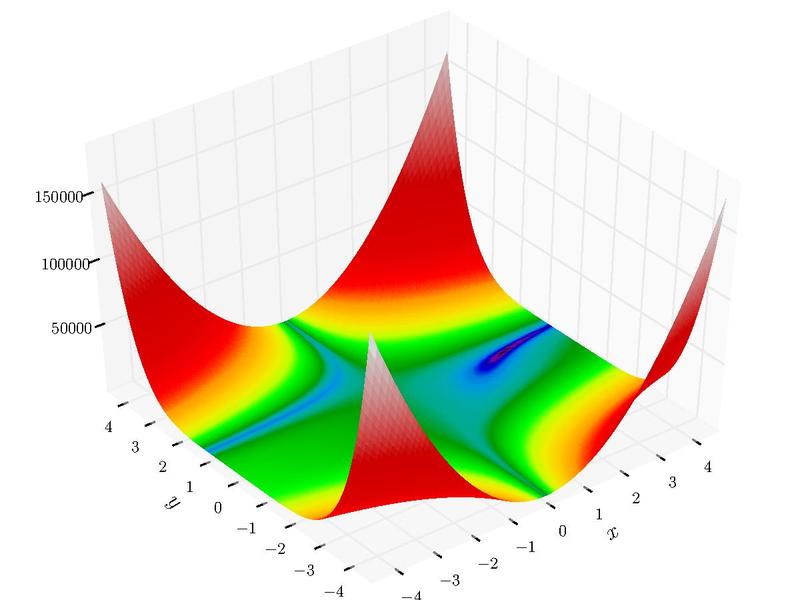
**Tabela 1.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Średnia ilość kroków | Średnia liczba wywołań funkcji celu | Ilość odnalezionego minimum globalnego | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 30 | 82 | 74 | 0,013 | 1,014 | 2,949 |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 25 | 381 | 81 | 0,005 | 0,930 | 3,054 |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 20 | 486 | 83 | 0,002 | 1,013 | 3,017 |
| Metoda Nelder'a-Mead'a | 29 | 63 | 80 | 0,006 | 1,012 | 3,033 |
| Metoda Nelder'a-Mead'a równolegle | 23 | 132 | 83 | 0,003 | 1,049 | 2,927 |

Każdy z przedstawionych algorytmów obliczył minimum tej funkcji testowej   
w małej ilości kroków – jest to spowodowane nie skomplikowaniem funkcji celu.  
Wszystkie algorytmy w dużej ilości testów odnalazły globalne minimum, ponieważ wzór funkcji był prosty dlatego wartości w każdym z algorytmów są bardzo zbliżone.

## Funkcja testowa Baley’s

Trzecia z kolei funkcja testowa przedstawiona w tej pracy to funkcja Beale’s posiadająca minimum w *f*( 3, 0.5 ) = 0. Graficzne przedstawienie badanej funkcji znajduje się na rysunku 16.



**Rys. 16.** Funkcja testowa : Beale's  
Źródło : [7]

Wzór funkcji:

|  |  |
| --- | --- |
|  | (9) |

Zestawienie wyników znajduje się w tabeli 2:

**Tabela 2.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Średnia ilość kroków | Średnia liczba wywołań funkcji celu | Ilość odnalezionego minimum globalnego | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 42 | 114 | 34 | 0,008 | 2,790 | 0,440 |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 38 | 559 | 64 | -0,003 | 3,035 | 0,498 |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 27 | 727 | 58 | 0,005 | 3,180 | 0,548 |
| Metoda Nelder'a-Mead'a | 39 | 97 | 41 | -0,008 | 2,880 | 0,453 |
| Metoda Nelder'a-Mead'a równolegle | 25 | 141 | 49 | -0,002 | 2,930 | 0,491 |

W tym przypadku najlepiej poradził sobie zrównoleglony algorytm Nelder’a – Mead’a który w najmniejszą ilość kroków podał najdokładniejszy wynik. Jest to spowodowane najlepiej obraną ścieżką tworzenia nowych simpleksów dla tego typu funkcji testowych.

## Funkcja testowa Ackley’s

Kolejnym przykładem jak został pokazany w tej pracy jest funkcja testowa Ackley’s,

której minimum jest w *f*( 0, 0 ) = 0. Jej wykres przedstawiony jest na rysunku 14:

|  |
| --- |
| **Rys. 17.** Funkcja testowa : Ackley's Źródło : [7] |

A jej wzór charakteryzuje się:

|  |  |
| --- | --- |
|  | (9) |

W Tabeli 3 zostały zaprezentowane wyniki algorytmów :

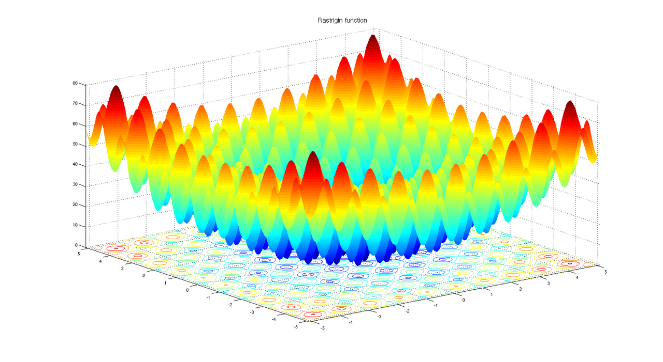
**Tabela 3.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Średnia ilość kroków | Średnia liczba wywołań funkcji celu | Ilość odnalezionego minimum globalnego | Wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 34 | 76 | 28 | -0,560 | 0,060 | -0,090 |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 28 | 441 | 58 | -0,280 | 0,040 | 0,100 |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 22 | 871 | 67 | -0,040 | 0,100 | 0,001 |
| Metoda Nelder'a-Mead'a | 37 | 101 | 32 | -0,310 | -0,030 | -0,120 |
| Metoda Nelder'a-Mead'a równolegle | 25 | 173 | 49 | -0,118 | 0,008 | 0,070 |

Ta funkcja testowa nie była skomplikowana i algorytmy obliczyły jej minimum w małej ilości iteracji. Każdy z algorytmów szybko poradził sobie z zadanym problemem, ale najdokładniejsze wartości przyniosły zrównoleglone algorytmy.

## Funkcja testowa Rastrigin 2D

Funkcja Rastrigin charakteryzuje się wieloma minimami lokalnymi. Z tego powodu często prawdziwe minimum znajdujące się w *f*(0, 0) = 0 nie zostaje odnalezione. Funkcja została przedstawiona na rysunku nr. 12:



**Rys. 18.** Funkcja testowa : Rastrigin  
Źródło : [7]

A jej wzór wygląda następująco:

|  |  |
| --- | --- |
|  | (8) |

Gdzie:  
*n*–liczba parametrów wejściowych

*xi-*współrzędne punktu

W Tabeli nr 4 zostały zestawione obliczenia dla każdego z algorytmów.

**Tabela 4.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Średnia ilość kroków | Średnia liczba wywołań funkcji celu | Ilość odnalezionego minimum globalnego | Średnia wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 65 | 182 | 7 | 0,990 | -0,990 | 0,990 |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 53 | 1263 | 23 | 0,990 | 0,990 | 1,600 |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 41 | 1834 | 39 | 0,009 | -0,002 | -0,006 |
| Metoda Nelder'a-Mead'a | 88 | 199 | 6 | 1,010 | -0,990 | -0,990 |
| Metoda Nelder'a-Mead'a równolegle | 65 | 480 | 21 | 0,010 | 0,003 | 0,006 |

Z powyższej tabeli można wywnioskować, że w przypadku funkcji Rastrigin nie zawsze zostaje znalezione minimum globalne. Algorytmy które w jednej iteracji liczyły tylko jeden punkt często nie odnalazły prawdziwego minimum globalnego. Jest to spowodowane tym, że nasza funkcja posiada wiele minimów lokalnych, a zadeklarowany najmniejszy błąd oraz ilość iteracji nie pozwala na większe przeszukiwanie przestrzeni. W tym przypadku najdokładniejsze wyniki przyniosła druga wersja równoległa algorytmu Hook’a–Jeeves’a.

## Funkcja testowa Rastrigin 5D

Funkcja testowa Rastrigin ze względu na swój charakter może być także testowana w przestrzeni wielowymiarowej, postanowiłem przetestować ją także w przestrzeni 5D.  
Aby obliczenia były dokładniejsze ilość iteracji została zwiększona do 250.

**Tabela 5.** Zestawienie wyników obliczeń algorytmów dla badanej funkcji testowej

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Średnia ilość kroków | Średnia liczba wywołań funkcji celu | Ilość odnalezionego minimum globalnego | Średnia wartość | x1 | x2 |
| Metoda Hook'a-Jeeves'a | 213 | 483 | 3 | 3,990 | -1,990 | 1,220 |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 177 | 6549 | 55 | 1,121 | 0,990 | 1,730 |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 169 | 12734 | 63 | 0,511 | -0,003 | -0,009 |
| Metoda Nelder'a-Mead'a | 238 | 455 | 1 | 4,490 | -0,013 | -1,98 |
| Metoda Nelder'a-Mead'a równolegle | 196 | 2843 | 43 | 1,310 | -0,003 | 0,001 |

Z tabeli 5 można wywnioskować, że funkcja w przestrzeni pięciowymiarowej była trudniejsza do rozwiązania i pochłonęła większą moc obliczeniową procesora. Zwiększenie ilości iteracji pozwoliło na określenie dokładniejszych wyników lecz w przypadku niezrównoleglonych wersji metod często pozostała już w ekstremum lokalnym. Współbieżne algorytmy lepiej poradziły sobie z tym problemem, dysponując mocniejszym procesorem z ośmioma wątkami   
i odpowiednio rozdzielić moc pomiędzy dostępnymi wątkami obliczenia były znacznie szybsze i dokładniejsze.

## Opracowanie wyników

Każdy z algorytmów w wersji podstawowej oraz zrównoleglonej poradził sobie z funkcjami testowymi. Można zauważyć, że wraz z zwiększoną ilością wywołań funkcji celu w pojedynczej iteracji wartość jest dokładniejsza oraz zostaje odnaleziona w mniejszej ilości kroków. Jednak najważniejszym powodem zrównoleglenia tych algorytmów, było zmniejszenie czasu obliczeń. W tabeli 5 zostały zestawione czasy wykonania poszczególnych algorytmów:

**Tabela 6.** Zestawienie czasów wykonania funkcji testowych

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Funkcja testowa Ackley's | Funkcja testowa Baley's | Funkcja testowa Booth's | Funkcja testowa Rastigin 2D | Funkcja testowa Rastigin 5D |
| Metoda Hook'a-Jeeves'a | 39 ms | 43 ms | 27 ms | 55 ms | 87 ms |
| Metoda Hook'a-Jeeves'a losowe ścieżki | 17 ms | 24 ms | 19 ms | 43 ms | 62 ms |
| Metoda Hook'a-Jeeves'a równolegle – skoki z odchyleniem | 14 ms | 23 ms | 20 ms | 37 ms | 57 ms |
| Metoda Nelder'a-Mead'a | 67 ms | 41 ms | 23 ms | 65 ms | 95 ms |
| Metoda Nelder'a-Mead'a równolegle | 29 ms | 26 ms | 15 ms | 46 ms | 63 ms |

Z powodu nieskomplikowanych funkcji testowych w przestrzeni dwuwymiarowej czasy algorytmów są bardzo niskie. Każda z wersji równoległych przyniosła skrócenie czasu obliczeń w pewnym stopniu. Z danych przedstawionych w Tabeli 6 można wywnioskować,  
że zrównoleglenie algorytmów metodami przedstawionymi w rozdziale czwartym przyniosło oczekiwane rezultaty. Punkty były przeliczane na 4 wątkach co w pewien sposób ograniczało obliczenia. W przypadku dostępności maszyny wielordzeniowej algorytmy wykorzystywałby wszystkie dostępne wątki i czasy obliczeń znacząco by spadły.

Na poniższym rysunku 16 został przedstawiony wykres zależności iteracji od minimalizacji wartości funkcji celu.

**Rys. 19.** Wykres zależności wartości funkcji celu w kolejnych iteracjach   
Źródło : opracowanie własne

Można zauważyć, że algorytmy zrównoleglone w wcześniejszym kroku zbliżyły się do wartości minimum globalnego. Bardzo ważne było pierwsze 10 kroków gdzie mieliśmy największy skok spadku wartości funkcji celu. Współbieżne algorytmy szybciej i częściej osiągnęły ekstremum globalne, a co za tym idzie dokładność wyników była większa.

# Podsumowanie

W pracy zostały przedstawione algorytmy, które z natury nie są równoległe. Żeby to zrobić trzeba zastanowić się jak powinno to być wykonane i czy przyniesie to oczekiwane skutki. Każdy pomysł zrównoleglenia algorytmów został starannie przemyślany oraz przetestowany, a wyniki tylko pokazały, że wybrana droga jest słuszna.   
Każdy z zrównoleglonych algorytmów przyniósł dokładniejsze obliczenia oraz skrócenie czasu obliczeń. Dzięki działaniu algorytmów na wielu wątkach przestrzeń przeszukiwań punktów była dużo większa co pozwoliło np. dla funkcji testowej Rastragin dużo częściej odnaleźć prawdziwe ekstremum funkcji. Metody obliczające tylko jeden punkt w jednej iteracji w dużej mierze odnajdywały tylko minimum lokalne, z którego już nie potrafiły się wydostać i kończyły obliczenia. Z tego powodu wartość funkcji celu była dużo większa niż prawdziwa.

W pracy zostało dowiedzione, że sposób w jaki metody zostały zrównoleglone i przy odpowiednim dobraniu współczynników, ilości punktów do przeliczenia na jeden wątek czas obliczeń może znacząco spaść. Na czas i dokładność obliczeń wpływ ma dostępny procesor na jakim będziemy testować lub wykonywać algorytmy. W zależności od ilości wątków do wykorzystania możemy tak rozłożyć obliczenia równolegle, aby osiągnąć jak najmniejszy czas obliczeń przy jednoczesnym uzyskaniu bardzo dokładnego wyniku.

Obliczenia równoległe w dzisiejszych czasach są dominującym wzorcem w architekturze komputerowej, stało się to za sprawą upowszechnienia procesorów wielordzeniowych.   
Ze względu na fizyczne ograniczenia uniemożliwiające dalsze zwiększanie częstotliwości taktowania procesorów środowiska masowo równoległe zaczęły powstawać coraz częściej. Algorytmy równoległe są bardzo wartościowe, wykonywanie w danej chwili wielu operacji jest cenne ze względu na możliwość szybszego obliczania różnego typu zagadnień.   
Metody przedstawione w tej pracy z łatwością można połączyć z serwerami np. Cyfronet AGH i wykorzystywać je m. in do obliczeń wielkoskalowych czy symulacji numerycznych.

# Literatura

[1] Vázquez S., Martín M.J., Fraguela B.B., Gómez A., Rodríguez A., Elvarsson B.B., *Novel parallelization of simulated annealing and Hooke & Jeeves search algorithms for multicore systems with application to complexfisheries stock assessment models*, Journal of Computational Science, 2016, in print

[2] Luchi F. Renato A. Krohling *Differential Evolution and Nelder – Mead for Constrained Non – linear Integer Optimization Problems*, Procedia Computer Science, Volume 55, 2015, Pages 668 - 677

[3] http://optymalizacja.w8.pl/simplexNM.html, maj 2008

[4] Nelder J. A., Mead R., *A simplex method for function minimization*, Computer Journal, Volume 7, 1965, 308-315

[5] Migallón H. Migallon V. Penades J., *Parallel alternating iterative algorithms with and without overlapping on multicore architectures*, Advances in Engineering Software, Volume 101, 2016, Pages 27-36

[6] Zhang Yu, Jian-Ping A. Chen P. *Research of Hybrid Programming with C#.Net and Matlab*, Physics Procedia, Volume 24, 2012, Pages 1677 – 1681

[7] https://en.wikipedia.org/wiki/Test\_functions\_for\_optimization