SPRAWOZDANIE Z PROJEKTU TEORII I METOD OPTYMALIZACJI

Rozwiązanie zadania programowania nieliniowego z wykorzystaniem metody pełzającego simpleksu Neldera-Meada

Prowadząca: dr inż. Ewa Szlachcic
Termin: poniedziałek TP $13^{15} - 15^{00}$

Kacper Połatajko 241603
Bartosz Kubiak 241611

1 czerwca 2021

1 Opis projektu

Celem projektu jest rozwiązanie zadania programowania nieliniowego przy pomocy metody pełzającego simpleksu Neldera-Meada. Zadanie polega na znalezieniu minimum funkcji $f(x)$ nieliniowej, ciągłej, bez ograniczeń dla $x \in R^n \ : \ n \leqslant 5$ oraz wymiarów kostki $a_i \leqslant x_i \leqslant b_i$, gdzie $i = 1, ..., n$.

2 Opis algorytmu

Podczas implementacji alogrytmu bazowano na opisach i rozwiązaniach zawartych w [2] i [3].
Metoda Neldera-Meada (zwana też metodą pełzającego simpleksu) jest bezgradientową metodą numeryczną stosowaną do wyznaczania minimum nieliniowych, n - wymiarowych funkcji. Przy wyznaczaniu minimum nie korzysta z pochodnych, dzięki czemu może zostać zastosowana do funkcji nieróżniczkowalnych. Jej działanie polega na kolejnym przekształcaniu simpleksu, posiadającego $n+1$ liniowo niezależnych wierzchołków, w kierunku szukanego minimum funkcji. W kolejnych iteracjach zmieniane jest położenie punktu simpleksu, w którym wartość funkcji celu jest najgorsza.

2.1 Szczegóły stosowanego algorytmu optymalizacji

Zastosowany algorytm jest algorytmem optymalizacji lokalnej. Za przyjęte kryteria stopu przyjęto maksymalną liczbę możliwych iteracji algorytmu oraz liczbę ε wyznaczającą dokładność największej odległości między dwoma wierzchołkami simpleksu. Zaimplementowany algorytm jest metodą minimalizacji bez ograniczeń.

2.2 Poszczególne kroki algorytmu

Algorytm korzysta ze zmiennych oraz operacji przedstawionych poniżej:

Zmienne:

- α - współczynnik odbicia ($\alpha > 0$),
- β - współczynnik kontrakcji ($0 < \beta < 1$),
- γ - współczynnik ekspansji ($\gamma > 0$),
- ϵ - dokładność obliczeń,
- L - liczba iteracji,
- $a_i \leqslant x_i \leqslant b_i$ dla $i = 1, ..., n$ - wymiary kostki,
- x_0 - punkt startowy,
- P_i - punkt i-ty simpleksu (i-ty wierzchołek) dla $i = 1, ..., n + 1$,
- P_{high} - punkt simpleksu, dla którego wartość funkcji celu jest największa,
- P_{low} - punkt simpleksu, dla którego wartość funkcji celu jest najmniejsza,
- P_{sym} - centroid simpleksu bez punktu P_{high}

Operacje na zmiennych:

- $P_{odb} = (1 + \alpha) \cdot P_{sym} - \alpha \cdot P_{high}$ - odbicie punktu P_{high} względem środka symetrii,
- $P_{eks} = (1 + \gamma) \cdot P_{odb} - \gamma \cdot P_{sym}$ - ekspansja punktu P_{odb} względem środka symetrii,
- $P_{kont} = \beta \cdot P_{high} + (1 - \beta) \cdot P_{sym}$ - kontrakcja punktu P_{high} względem środka symetrii,
- $P_i = P_i + P_{low}/2$ dla $i = 1, ..., n + 1$ - operacja redukcji.

Działanie algorytmu zostało przedstawione poniżej:

1. Stworzenie simpleksu z losowo wybranego punktu startowego, ograniczonego zakresami kostki, o $n + 1$ wierzchołkach.

2. Obliczenie wartości funkcji w wierzchołkach simpleksu.

3. Wyznaczenie punktów simpleksu, dla których wartość funkcji celu jest największa F_{max} i najmniejsza F_{min}.

4. Obliczenie punktu stanowiącego środek symetrii (z wyłączeniem punktu P_{high}) i wartości funkcji celu w tym punkcie.

5. Wykorzystanie operacji odbicia punktu P_{high} względem środka symetrii i wyznaczenie jego wartości funkcji celu F_{odb}.

6. Jeśli wartość $F_{odb} < F_{min}$ to:

   (a) Wykonanie operacji ekspansji punktu odbicia P_{odb} względem środka symetrii i wyznaczenie wartości funkcji celu F_{eks} w nowym punkcie.

   - jeśli wartość $F_{eks} < F_{min}$ to za obecny punkt P_{high} podstawiamy punkt ekspansji P_{eks},
   - jeśli wartość $F_{eks} \geqslant F_{min}$ to za obecny punkt P_{high} podstawiamy punkt odbicia P_{odb}

   (b) Sprawdzamy kryteria stopu. Jeśli nie jest spełnione któreś z kryteriów stopu, to wracamy do punktu 2.

7. Jeśli wartość $F_{odb} > F_{min}$ to:

   (a) Jeżeli wartość F_{odb} jest większa, bądź równa wartości funkcji celu w każdym punkcie simpleksu (oprócz punktu P_{high}) to:

   - jeśli $F_{odb} \geqslant F_{max}$, to przechodzimy do kroku następnego,
   - jeśli $F_{odb} < F_{max}$, to za punkt P_{high} podstawiamy punkt odbicia P_{odb}

   (b) Wykonanie operacji kontrakcji punktu P_{high} względem środka symetrii i wyznaczenie wartości funkcji celu F_{kont} w nowym punkcie.

   - jeśli $F_{kont} \geqslant F_{max}$ wykonujemy operację redukcji bieżącego simpleksu,
   - jeśli $F_{kont} < F_{max}$, to za punkt P_{high} podstawiamy punkt kontrakcji P_{kont} i przechodzimy do ostatniego kroku

   (c) Jeżeli wartość F_{odb} jest mniejsza od wartości funkcji celu w każdym punkcie simpleksu (oprócz punktu P_{high}), to za punkt P_{high} podstawiamy punkt odbicia P_{odb}.

   (d) Sprawdzamy kryteria stopu. Jeśli nie jest spełnione któreś z kryteriów stopu, to wracamy do punktu 2.

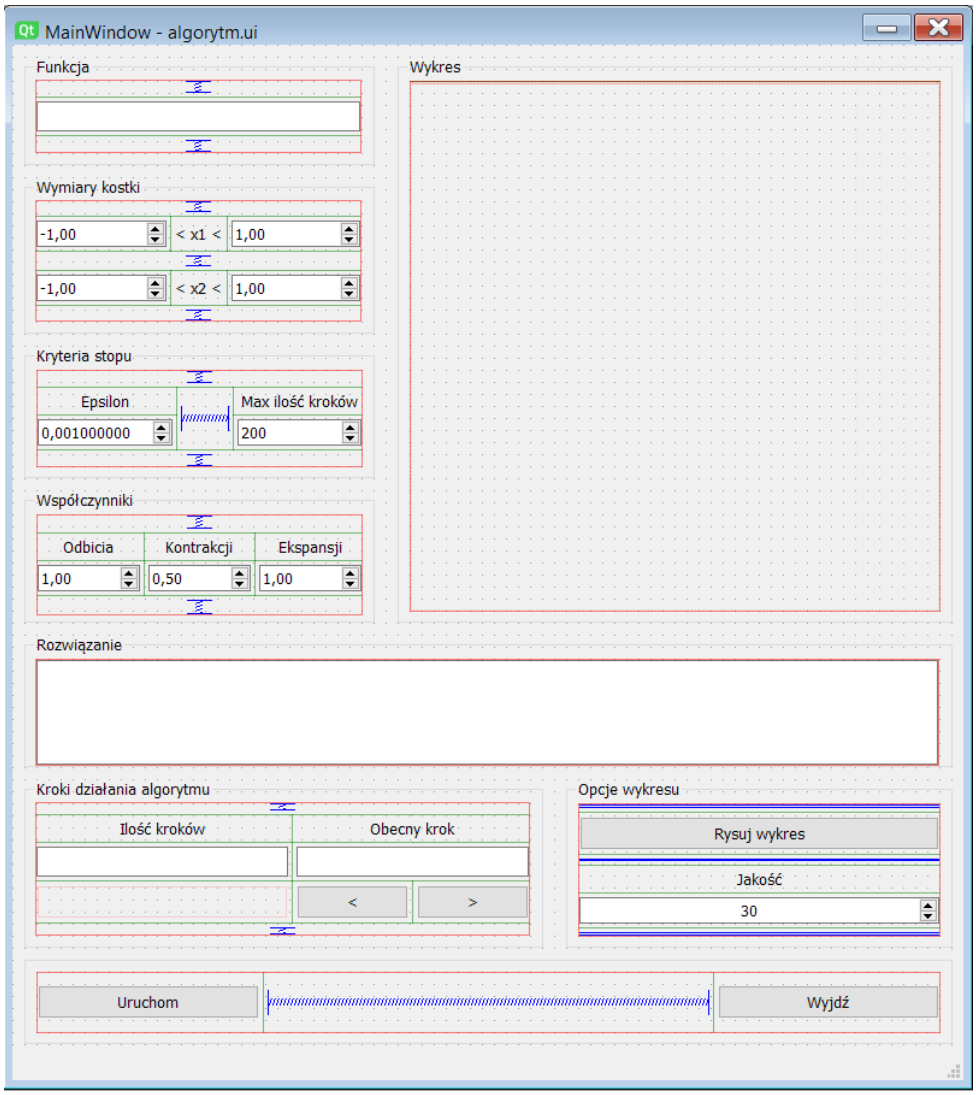
3 Implementacja programu

3.1 Funkcje i biblioteki

Program został napisany w języku *Python* przy wykorzystaniu nakładki *PyQt5* na bibliotekę *Qt* umożliwiającej stworzenie interfejsu graficznego. Operacje arytmetyczno-logiczne obsługiwane są przy pomocy funkcji *math* i *numpy*. Biblioteka *matplotlib* jest wykorzystywana do wizualizacji poszczególnych kroków algorytmu. Dodatkowo do działania programu wykorzystywane są funkcje:

- *eval* - sprawdzanie wymiaru zadania,
- *random* - losowanie wierzchołków simpleksu,
- *sys* - obsługa wywołań systemowych.

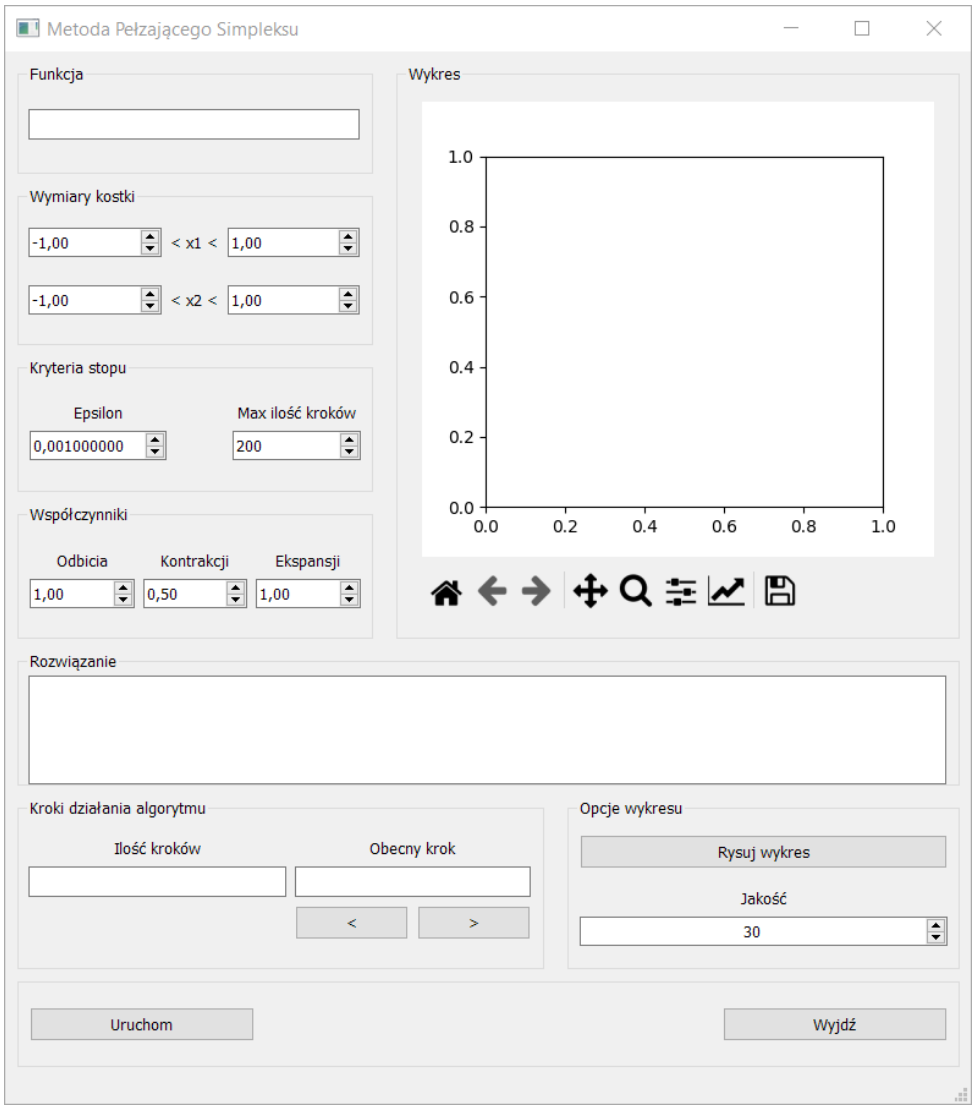
Program podzielono na część odpowiedzialną za inicjalizację graficzną okna i jego obsługę (*glowny.py*), część wyświetlającą wykresy warstwic i simpleksu (*wykres_ widget.py*) oraz część zawierającą implementację metody Neldera-Meada (*metodaNM.py*). W skład programu wchodzi też *algorytm.ui*, który jest plikiem wynikowym programu *Qt Designer*, w którym zaprojektowano wygląd okna aplikacji (rys. 1).



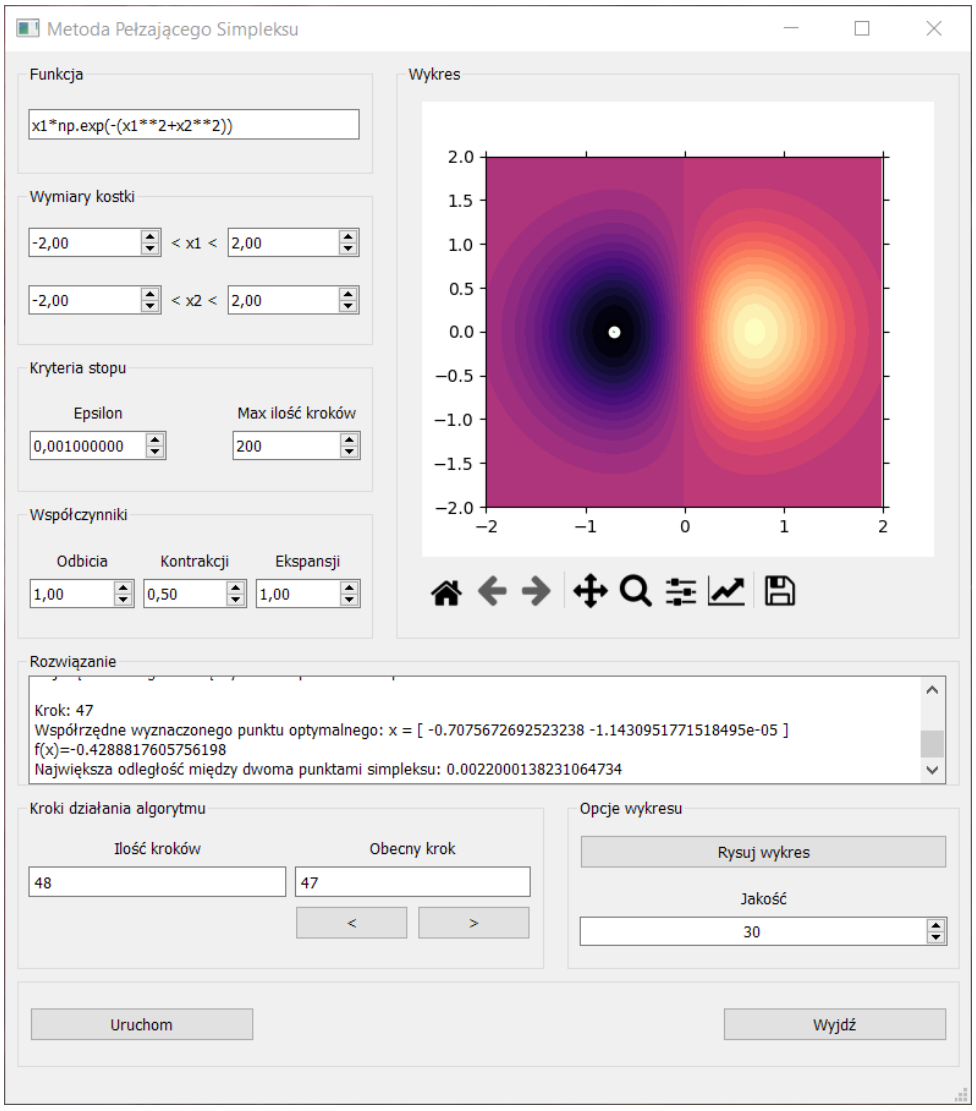
Rysunek 1: Projekt okna aplikacji w programie *Qt Designer*

3.2 Interfejs graficzny

Gotowy interfejs graficzny przedstawiono na rys. 2. Opis poszczególnych części okna przedstawiono poniżej.



Rysunek 2: Wygląd okna aplikacji zaraz po jego uruchomieniu



Rysunek 3: Wygląd okna aplikacji po uruchomieniu algorytmu dla przykładu funkcji z eksponentą

1. *Funkcja* - jest to pole, w którym podaje się funkcję do algorytmu,
2. *Wymiary kostki* - pole wyboru zakresu wymiarów kostki. Na podstawie tych wartości będzie też losowany punkt startowy algorytmu,
3. *Kryteria stopu* - umożliwia wybranie wartości dokładności ε oraz maksymalnej ilości kroków,
4. *Współczynniki* - zawiera pola, w których można wpisać wartości współczynników odbicia, kontrakcji i ekspansji. Bazowo mają one odpowiednio wartości 1, $0,5$ i 1,
5. *Rozwiązanie* - jest to pole, w którym algorytm wyświetla osiągnięte końcowe rozwiązanie (przykład na rys. 4) lub rozwiązanie dla danego kroku (przykład na rys. 5).

6. *Kroki działania algorytmu*:

   - pole *Ilość kroków* wyświetla ile iteracji wykonał algorytm zanim zatrzymał się po osiągnięciu danego kryterium stopu,
   - strefa *Obecny krok* umożliwia przejście po każdym wykonanym przez algorytm kroku, łącznie z jego wizualizacją. Odbywa się to przyciskami $<$ oraz $>$, przy czym numer obecnego kroku wyświetla się powyżej strzałek,

7. *Opcje wykresu*:

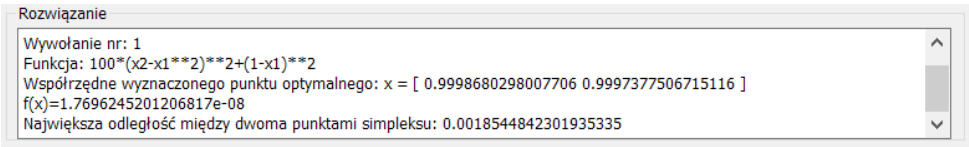
   - pole *Jakość* pozwala na zmianę ilości warstwic wykresu w zakresie 1 do 1000 (bazowo 30),
   - przycisk *Rysuj wykres* pozwala na wprowadzenie zmian (ilość warstwic oraz wyświetlany zakres) na wykresie bez ponownego uruchamiania algorytmu,

8. *Wykres* - ta część podzielona jest na dwa odrębne elementy:

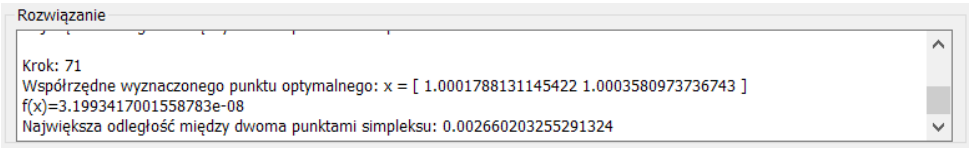
   - Obiekt *Canvas* - pozwala na wyrysowanie i wyświetlenie warstwic z danym rozwiązaniem,
   - Obiekt *Toolbar* - pozwala na manipulację wykresem (przybliżenie, oddalenie, przesuwanie, modyfikowanie osi oraz zapis bieżącego wykresu do pliku),

9. *Przyciski na samym dole*:

   - *Uruchom* - inicjalizuje działanie algorytmu dla wybranych parametrów oraz funkcji, wyświetla końcowy wynik w oknie *Rozwiązanie* oraz wyrysowuje końcowy wynik działania funkcji w oknie *Wykres*,
   - *Wyjdź* - całkowite zamknięcie programu i okna aplikacji,



Rysunek 4: Zawartość okna *Rozwiązanie* dla końcowego rozwiązania przykładu



Rysunek 5: Zawartość okna *Rozwiązanie* dla danego kroku przykładu

3.3 Sposób działania zaimplementowanego algorytmu

Algorytm działa na podstawie schematu obliczeń z podrozdziału 2.2. Cała metoda realizowana jest w jednej funkcji *algorytm()*, która działa w nieskończonej pętli *while*, której przerwanie realizowane jest poprzez osiągnięcie jednego z kryteriów stopu. Poszczególne operacje opisane w podrozdziale 2.2 mają swoje osobne funkcje, którą są wywoływane przez algorytm w odpowiednich momentach. Na końcu działania algorytmu (po osiągnięciu kryterium stopu) zwracane są wartości zmiennych, dla których funkcja osiąga obliczone minimum, liczbę kroków, w których osiągnięto minimum oraz wartość funkcji celu.

Poszczególne kroki działania przedstawiono poniżej:

1. Na samym początku działania algorytmu losowane są wierzchołki startowego simpleksu na podstawie podanych wcześniej wymiarów kostki (liczba wierzchołków to $n + 1$, gdzie n oznacza wymiar zadania).

2. Program sprawdza w tym momencie czy odległości między poszczególnymi wierzchołkami nie są mniejsze niż podana wartość ε. Wyklucza to błąd przy losowaniu początkowego simpleksu, gdy to punkty zostaną wylosowane bardzo blisko siebie i w dodatku w okolicy minimum.

3. W tym momencie funkcja wchodzi w nieskończoną pętlę *while*, z której może tylko wyjść poprzez osiągnięcie kryterium stopu.

4. Na początku pętli wyliczana jest wartość funkcji celu we wszystkich wierzchołkach i wyłaniane są te o wartości największej i najmniejszej.

5. Następnie wyznaczany jest środek symetrii simpleksu bez uwzględnienia punktu o największej funkcji celu i na koniec wyznaczana jest wartość funkcji celu w punkcie środka symetrii.

6. W tym momencie następuje odbicie wierzchołka simpleksu, o największej wartości funkcji celu, względem wyliczonego środka symetrii. Wyliczana jest wartość funkcji w tym punkcie.

7. Następnie sprawdzamy czy wartość funkcji w odbitym punkcie jest mniejsza, bądź większa od najmniejszej wartości w wierzchołku simpleksu.

8. Jeśli jest mniejsza:

   (a) Liczymy ekspansję odbitego punktu względem wyznaczonego środka symetrii. Jeśli wartość funkcji w wyznaczonym punkcie jest mniejsza od najmniejszej wartości funkcji w wierzchołku, to za punkt simpleksu o największej wartości funkcji podstawiamy punkt ekspansji.
   Natomiast jeśli wartość ta jest równa lub większa - za ten punkt podstawiamy jego odbicie.

(b) W tym momencie sprawdzane jest czy długość największej odległości między wierzchołkami simpleksu jest mniejsza niż zakładana wartość ε (kryterium stopu). Jeśli tak, to program przechodzi do punktu 11. Jeśli nie, to wracamy do punktu 4.

9. Jeśli jest większa:

   (a) Sprawdzamy czy wartość funkcji w obitym punkcie jest większa lub równa którejś z wartości funkcji w wierzchołkach simpleksu (bez wierzchołka z wartością największą). Jeśli nie to przechodzimy do kolejnego punktu. Jeśli tak i wartość ta jest większa od wartości największej dla pominiętego wierzchołka simpleksu, to za niego podstawiamy punkt odbicia.

   (b) Następnie wykonywana jest kontrakcja wierzchołka o największej wartości funkcji, względem środka symetrii. Jeśli otrzymana wartość funkcji w tym obliczonym punkcie jest większa lub równa największej wartości funkcji w wierzchołku, dokonujemy operacji redukcji na simpleksie i przechodzimy do sprawdzenia kryterium stopu w punkcie 9d.
   Jeśli jest mniejsza to za wierzchołek o największej wartości funkcji podstawiamy punkt kontrakcji i przechodzimy do kolejnego punktu.

   (c) Sprawdzamy czy wartość funkcji w odbitym punkcie jest mniejsza od wartości w wierzchołkach (bez punktu z największą wartością). Jeżeli tak to za wierzchołek z największą wartością funkcji podstawiamy punkt odbicia.

   (d) W tym momencie sprawdzane jest czy długość największej odległości między wierzchołkami simpleksu jest mniejsza niż zakładana wartość ε (kryterium stopu). Jeśli tak, to program przechodzi do punktu 11. Jeśli nie, to wracamy do punktu 4.

10. Dodatkowo sprawdzana jest możliwość kiedy to wartość funkcji w odbitym punkcie jest równa najmniejszej wartości w wierzchołku simpleksu. Wyświetlany jest wtedy komunikat na terminalu o błędzie. Metoda Neldera-Meada nie rozpatruje takiego przypadku, osiągnięcie takiej wartości najczęściej wiąże się z próbą rozwiązywania problemu liniowego metodą nieliniową.

11. Na koniec sprawdzane jest czy liczba iteracji nie osiągnęła już maksimum (kryterium stopu). Jeśli nie to wracamy do punktu 4. Jeśli tak to algorytm kończy działanie i zwracane są współrzędne punktu, dla którego osiągnięto minimum, wartość funkcji celu w tym punkcie oraz liczba wykonanych iteracji.

4 Działanie programu i wykonane testy

4.1 Ilustracja kolejnych iteracji na przykładzie

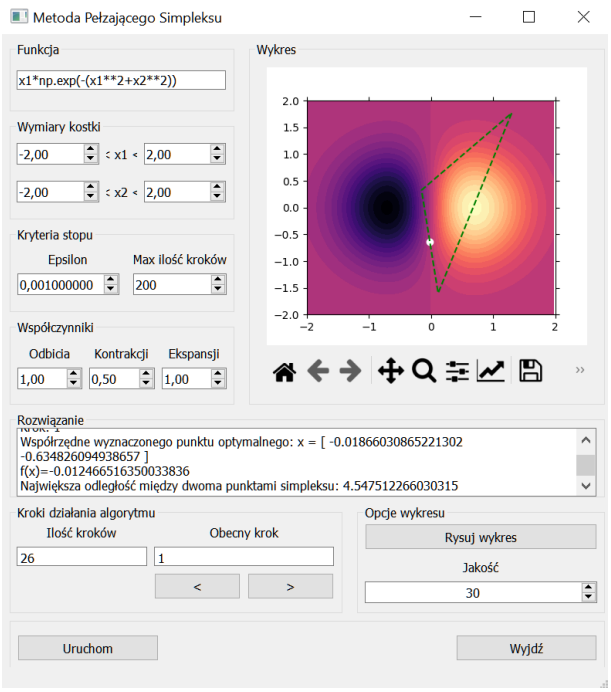
Nasze testy przeprowadziliśmy dla kilku przykładowych funkcji[1]. Kolejne kroki programu przedstawiliśmy dla przykładu funkcji z eksponentą, która posiada jedno minimum i jedno maksimum:

$$f(x) = x_1 \cdot exp(-(x_1^2 + x_2^2))$$

Wymiary kostki:

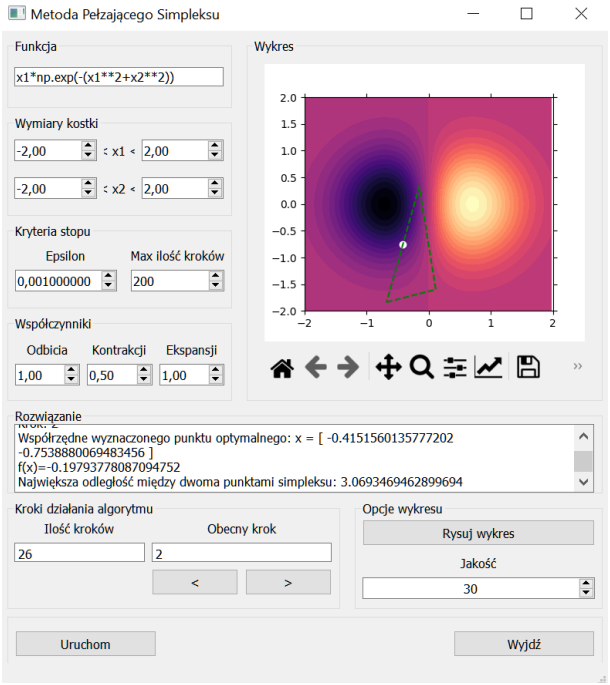
$$-2 \leqslant x_1 \leqslant 2$$
$$-2 \leqslant x_2 \leqslant 2$$

Krok 1:



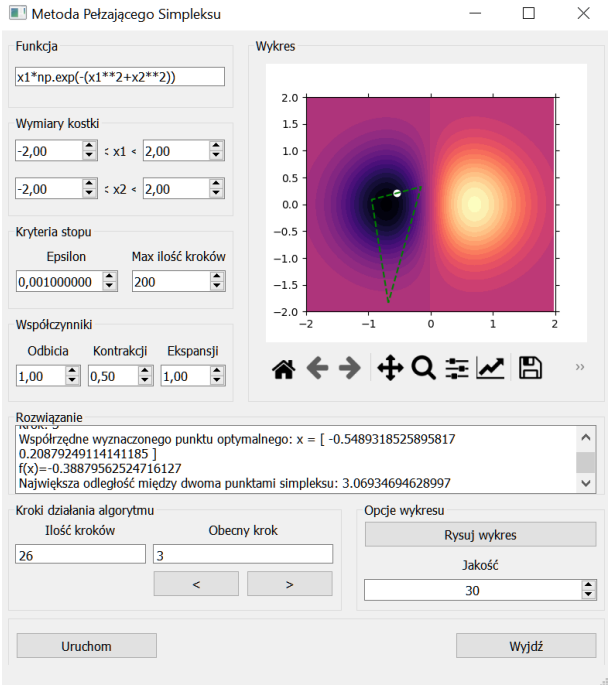
Rysunek 6: Krok pierwszy działania programu.

Krok 2:



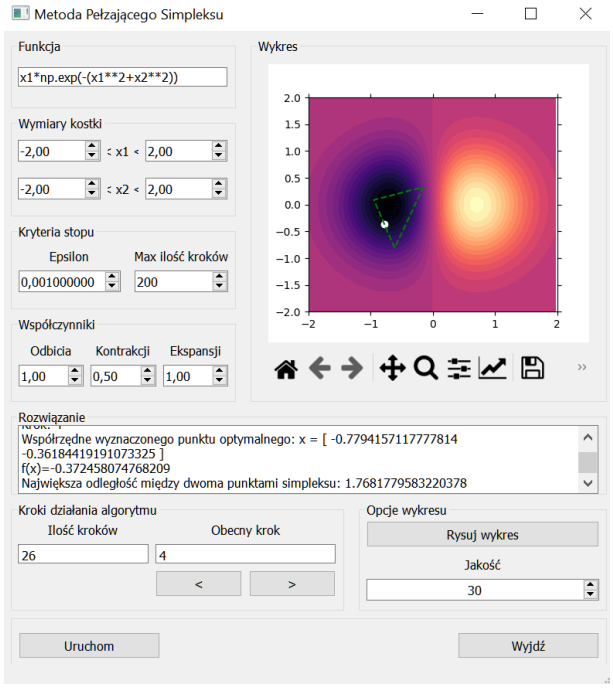
Rysunek 7: Krok drugi działania programu.

Krok 3:



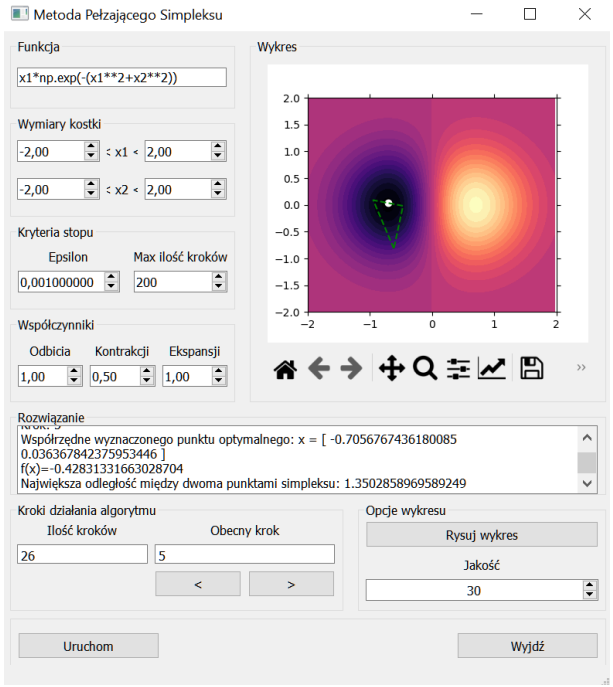
Rysunek 8: Krok trzeci działania programu.

Krok 4:



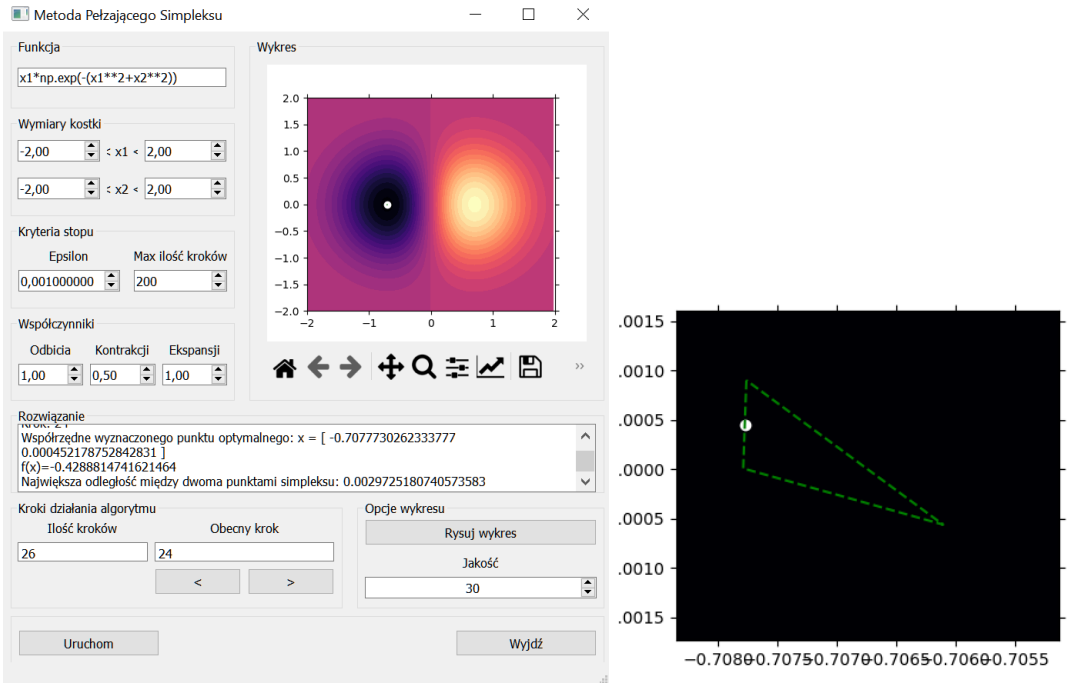
Rysunek 9: Krok czwarty działania programu.

Krok 5:



Rysunek 10: Krok piąty działania programu.

Krok 24:

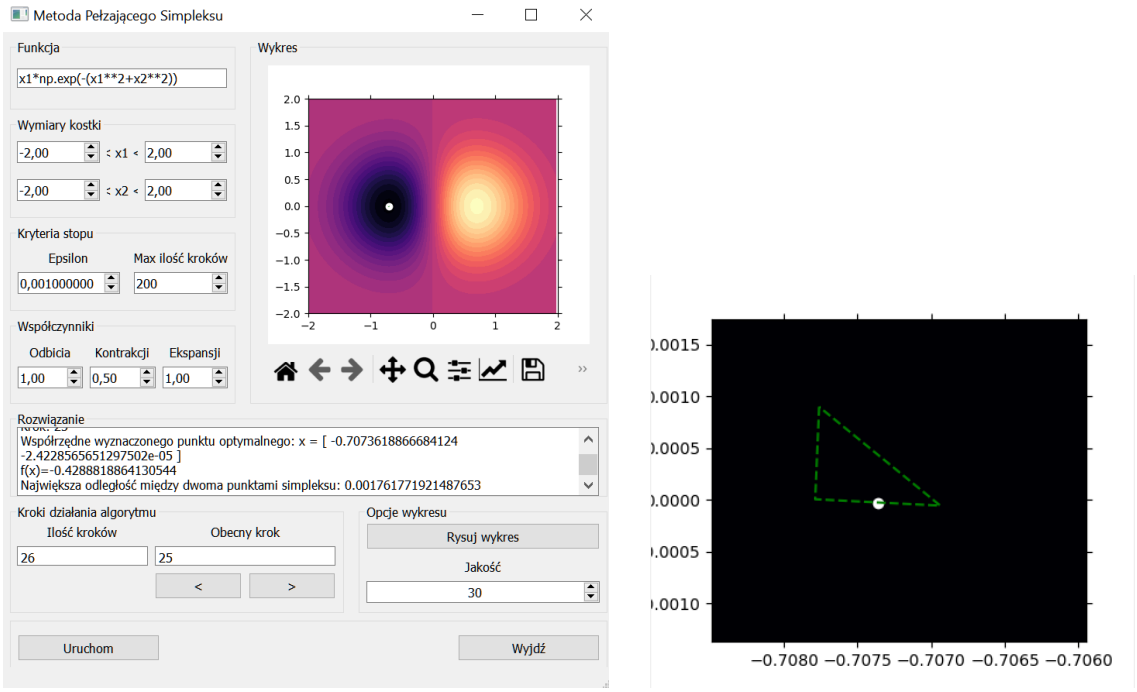


(a) krok 24

(b) przybliżenie wykresu krok 24

Rysunek 11: Krok dwudziesty czwarty działania programu.

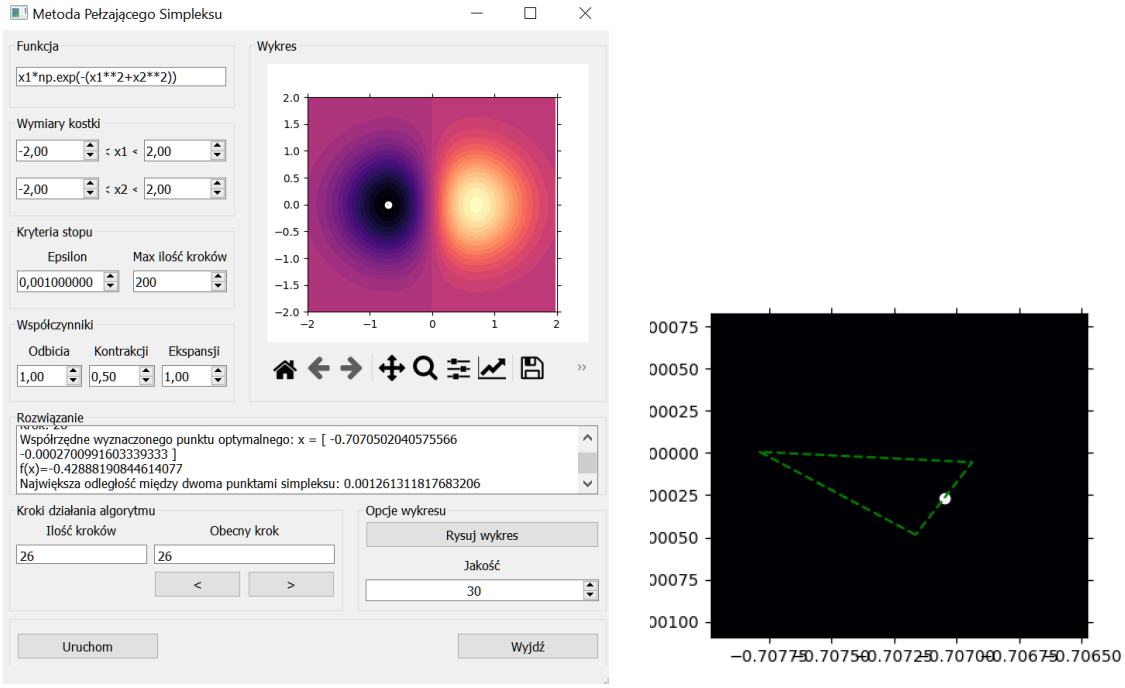
Krok 25:



(a) krok 25 (b) przybliżenie wykresu krok 25

Rysunek 12: Krok dwudziesty piąty działania programu.

Krok 26:



(a) krok 26

(b) przybliżenie wykresu krok 26

Rysunek 13: Krok dwudziesty szósty działania programu.

Na powyższych rysunkach (5-12) widać skrócone działanie programu krok po kroku. Można na nich zauważyć pełzanie simpleksu i znajdowanie przez algorytm nowych punktów simpleksu w stronę punktu z minimum funkcji. Wyliczona przez algorytm wartość minimalna podanej funkcji to:

$$f(x^*) = -0.42888190844614077$$

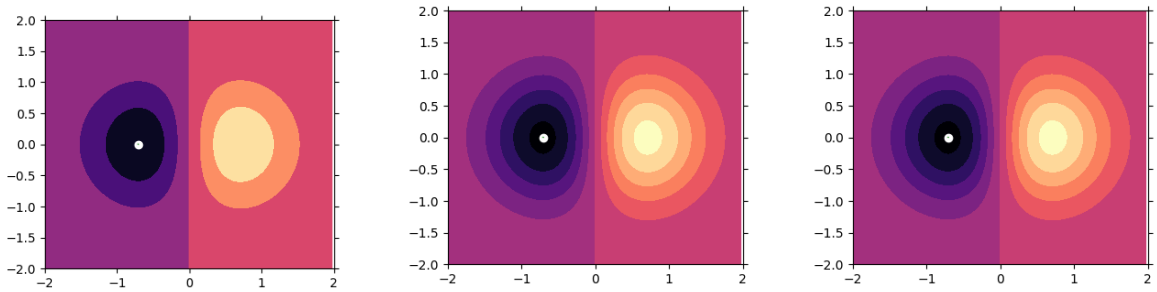
w punkcie:

$$x^* = [-0.7070502040575566 \ \ -0.0002700991603339333]$$

co jest zbliżone do oczekiwanych wartości.

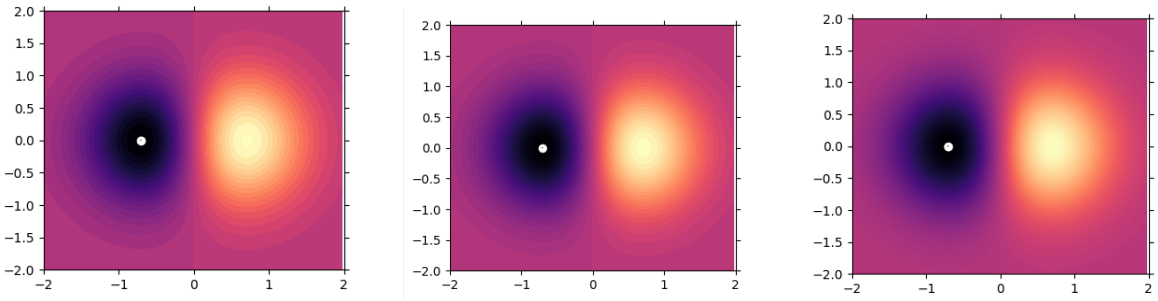
4.2 Różnica w ilości warstwic

Kolejnym testem jaki przeprowadziliśmy dla naszego programu było wyrysowanie wykresu w zależności od ilości warstwic (zmiana paramteru jakości w programie).



(a) 5 warstwic (b) 10 warstwic (c) 20 warstwic

Rysunek 14: Różnice w ilości warstwic.



(a) 35 warstwic (b) 50 warstwic (c) 100 warstwic

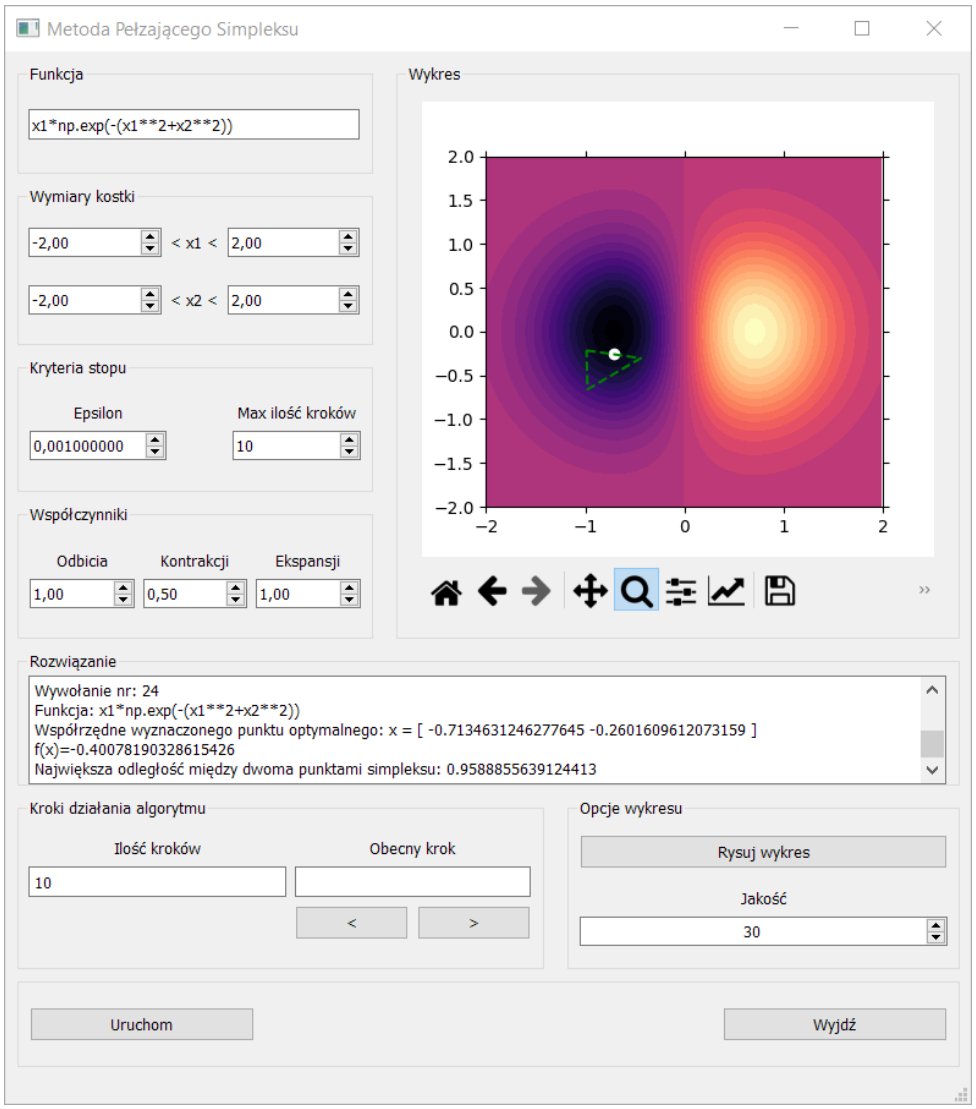
Rysunek 15: Różnice w ilości warstwic.

Zmiana ilości warstwic wpływała jedynie na wygląd wykresu. Zmniejszenie lub zwiększenie jakości nie miało skutków w dokładności otrzymanego wyniku.

4.3 Wpływ kryteriów stopu i współczynników na wynik

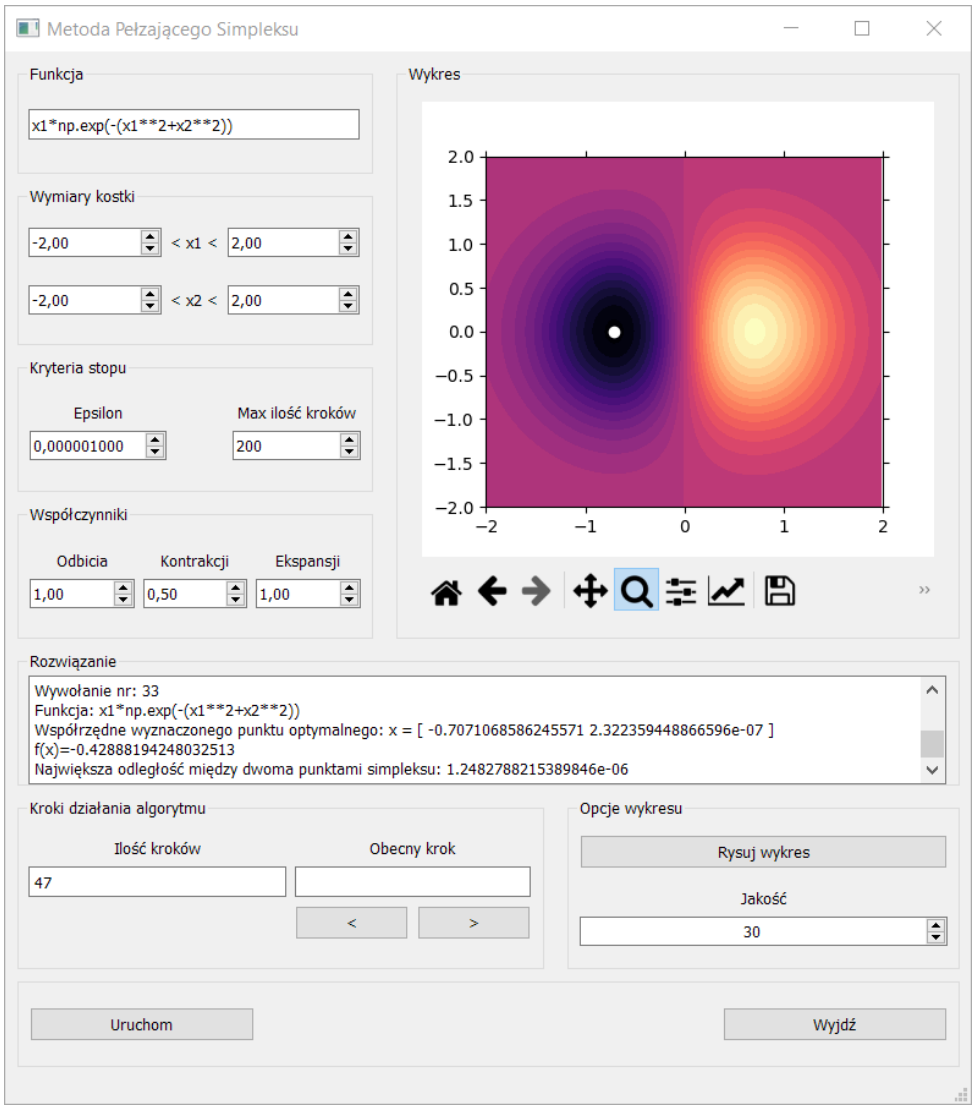
Na dokładność otrzymanego wyniku możemy wpływać poprzez wartość kryteriów stopu oraz wartości współczynników odbicia, kontrakcji i ekspansji. Ich wpływ przedstawiono poniżej.

Kryteria stopu mają znaczny wpływ na otrzymywany wynik. Przy za małej ilości kroków algorytm nie zdąży dojść do minimum i będzie musiał się przedwcześnie zatrzymać. Za duża wartość kroku może prowadzić do niepotrzebnego wydłużania czasu obliczeń (najczęściej ograniczony jest wcześniej przez drugie kryterium stopu). Wpływ wartości maksymalnej liczby kroków na algorytm pokazano poniżej.

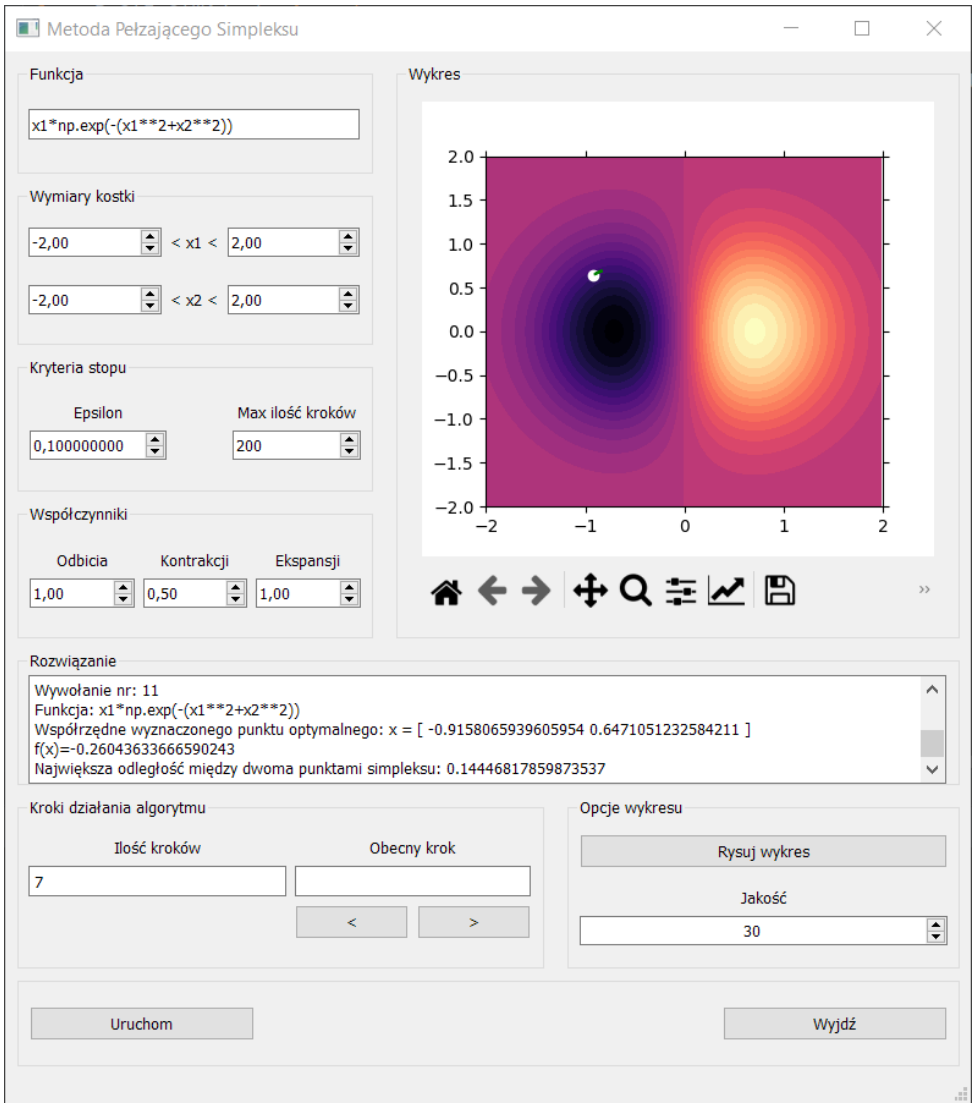


Rysunek 16: Wpływ za małej wartości maksymalnej liczby kroków na wynik

Zmiana wartości ε ma największy wpływ na otrzymywany wynik. Wyznacza on dokładność z jaką zostanie uzyskany końcowy wynik. Zbyt duża wartość prowadzi do wykonania większej ilości kroków, niepotrzebnie wydłużając tym samym czas działania programu. Zbyt mała wartość prowadzi do otrzymania wyników już w kilku-kilkunastu krokach, ale z bardzo małą dokładnością. Wyniki badań przedstawiono poniżej.

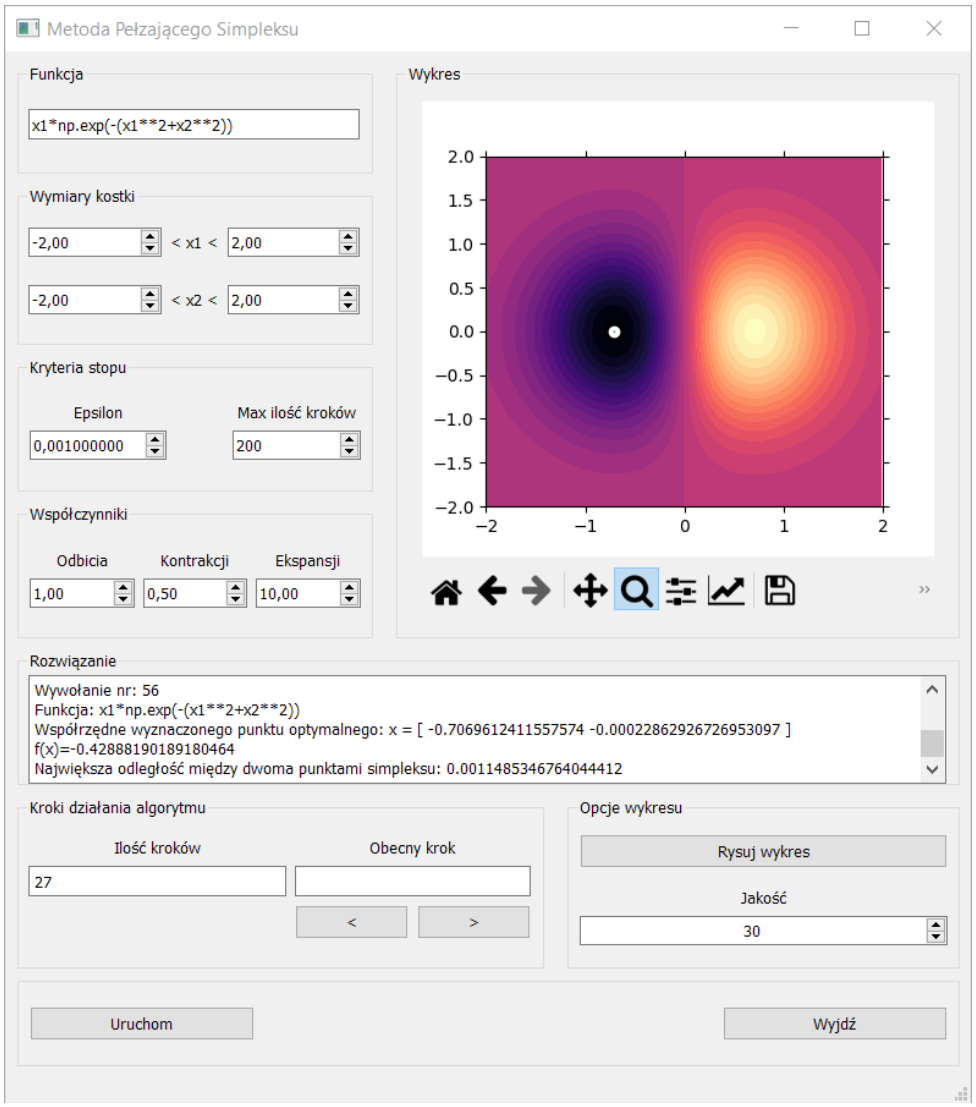


Rysunek 17: Wpływ za małej wartości ε na otrzymany wynik

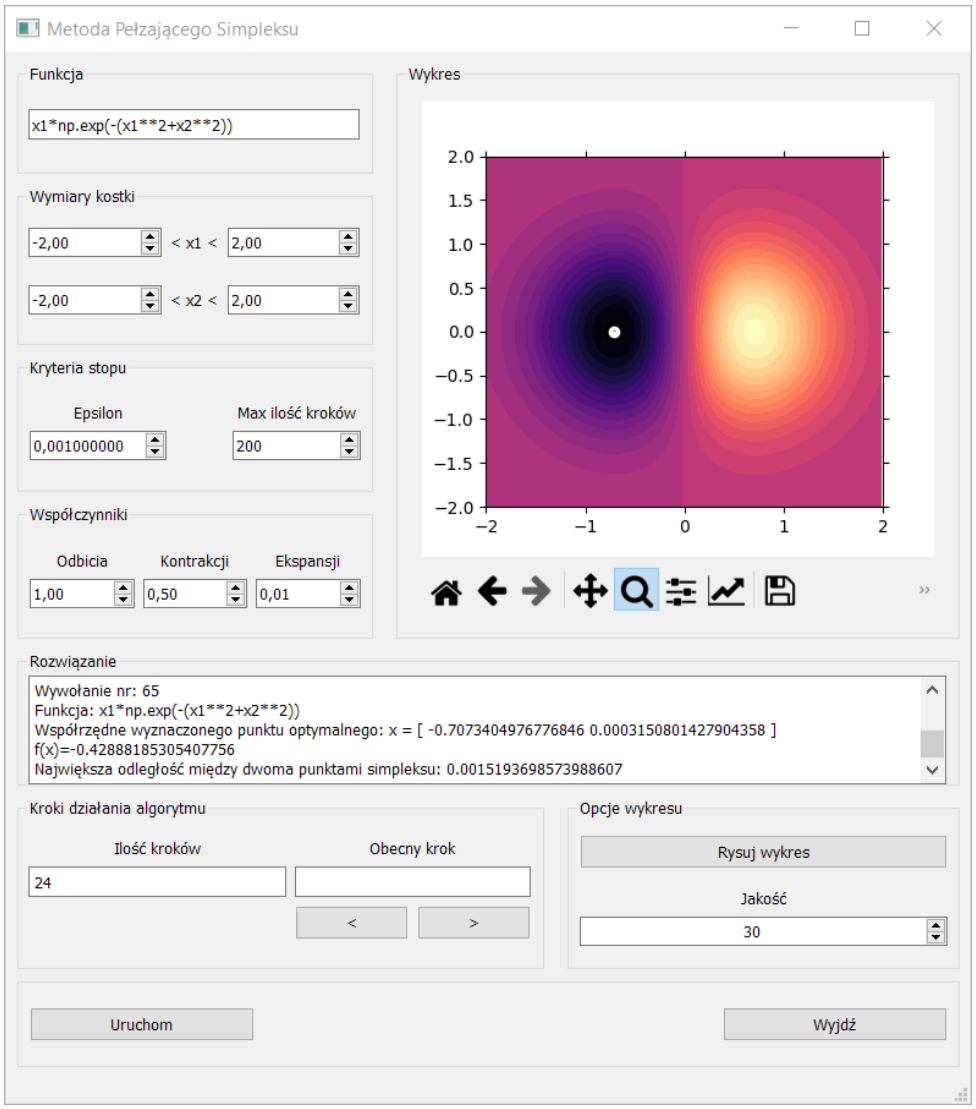


Rysunek 18: Wpływ za dużej wartości ε na otrzymany wynik

Nie tylko kryteria stopu mają wpływ na otrzymany wynik. Zmieniając parametry współczynników odbicia, kontrakcji i ekspansji wpływamy bezpośrednio na dokładność otrzymanego wyniku. Testy przedstawiono poniżej
Współczynnik ekspansji ma znikomy wpływ na dokładność otrzymanych wyników zarówno dla swojej dużej jak i małej wartości. Wykonane testy przedstawiono poniżej.

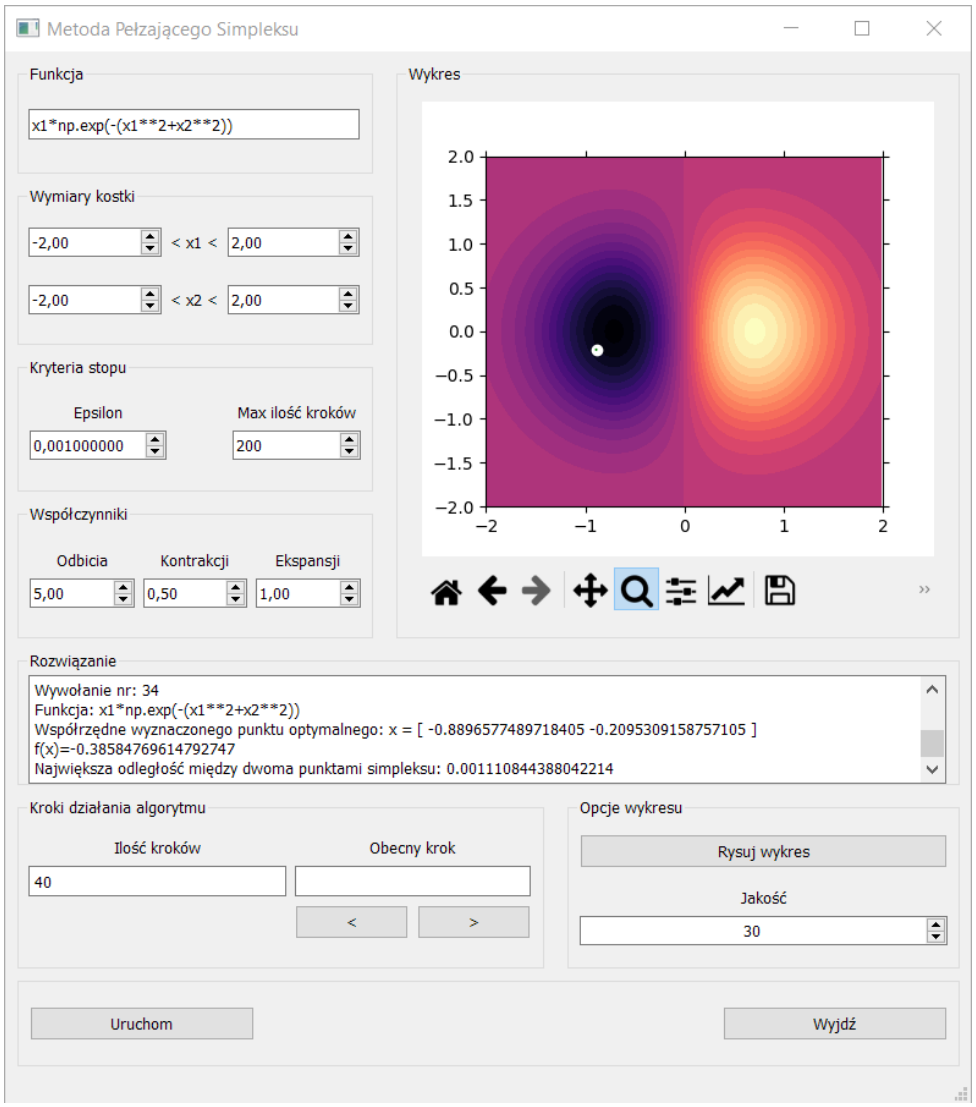


Rysunek 19: Wpływ dużej wartości współczynnika ekspansji na otrzymany wynik

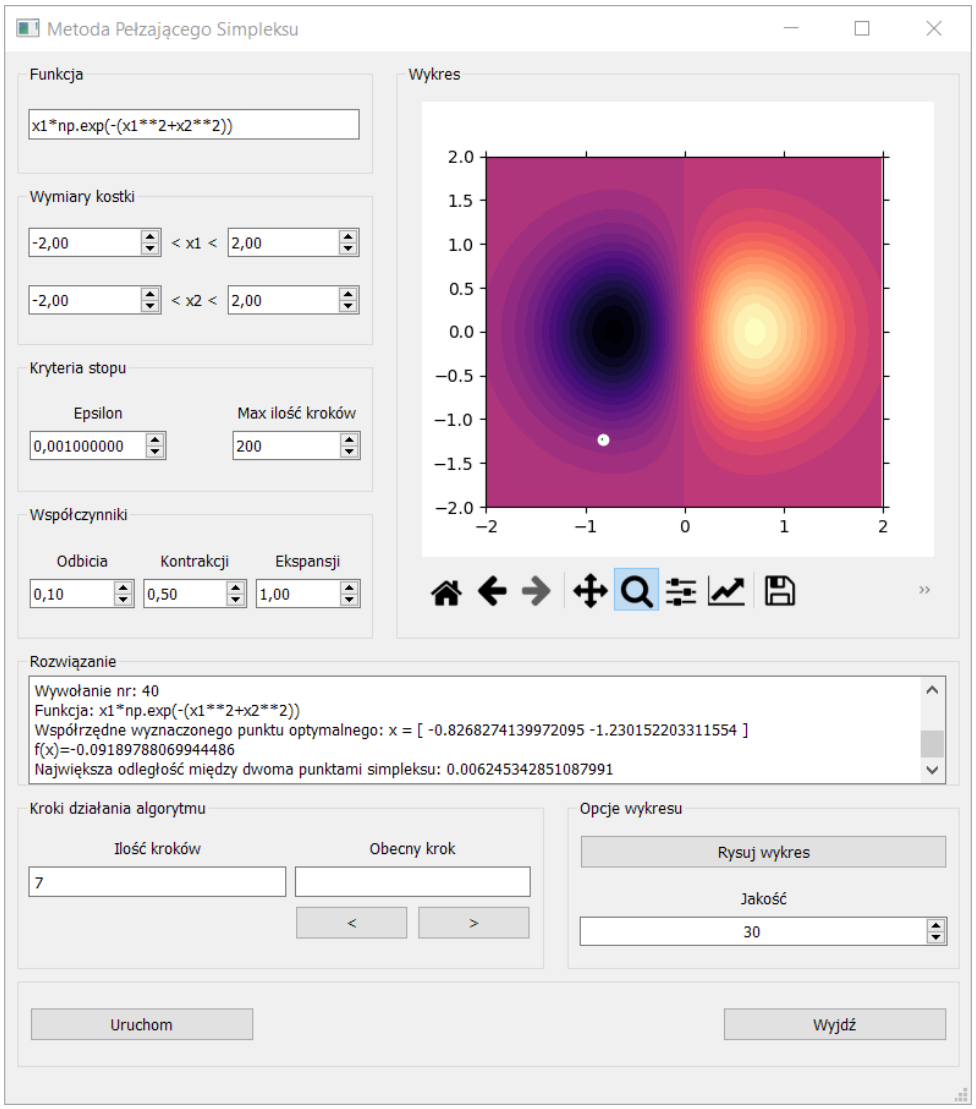


Rysunek 20: Wpływ małej wartości współczynnika ekspansji na otrzymany wynik

Współczynnik odbicia ma największy wpływ na wynik spośród pozostałych współczynników. Za mała wartość doprowadzi do zbyt szybkiego zatrzymania algorytmu i uniemożliwi dojście do minimum funkcji. Zaś zbyt duża wartość nie ograniczy liczby kroków, ale również nie pozwoli dojść do konkretnej wartości minimum. Wyniki testów przedstawiono poniżej.

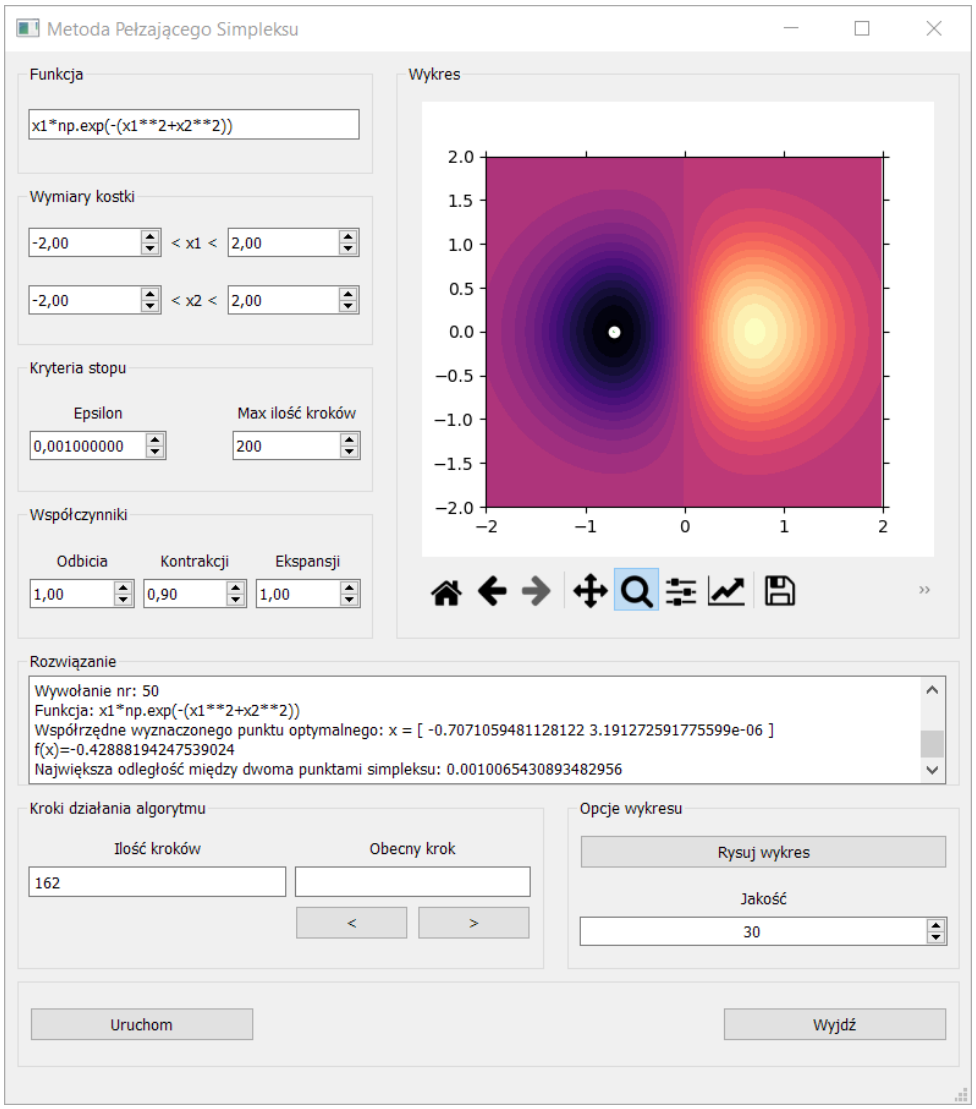


Rysunek 21: Wpływ za dużej wartości współczynnika odbicia na otrzymany wynik

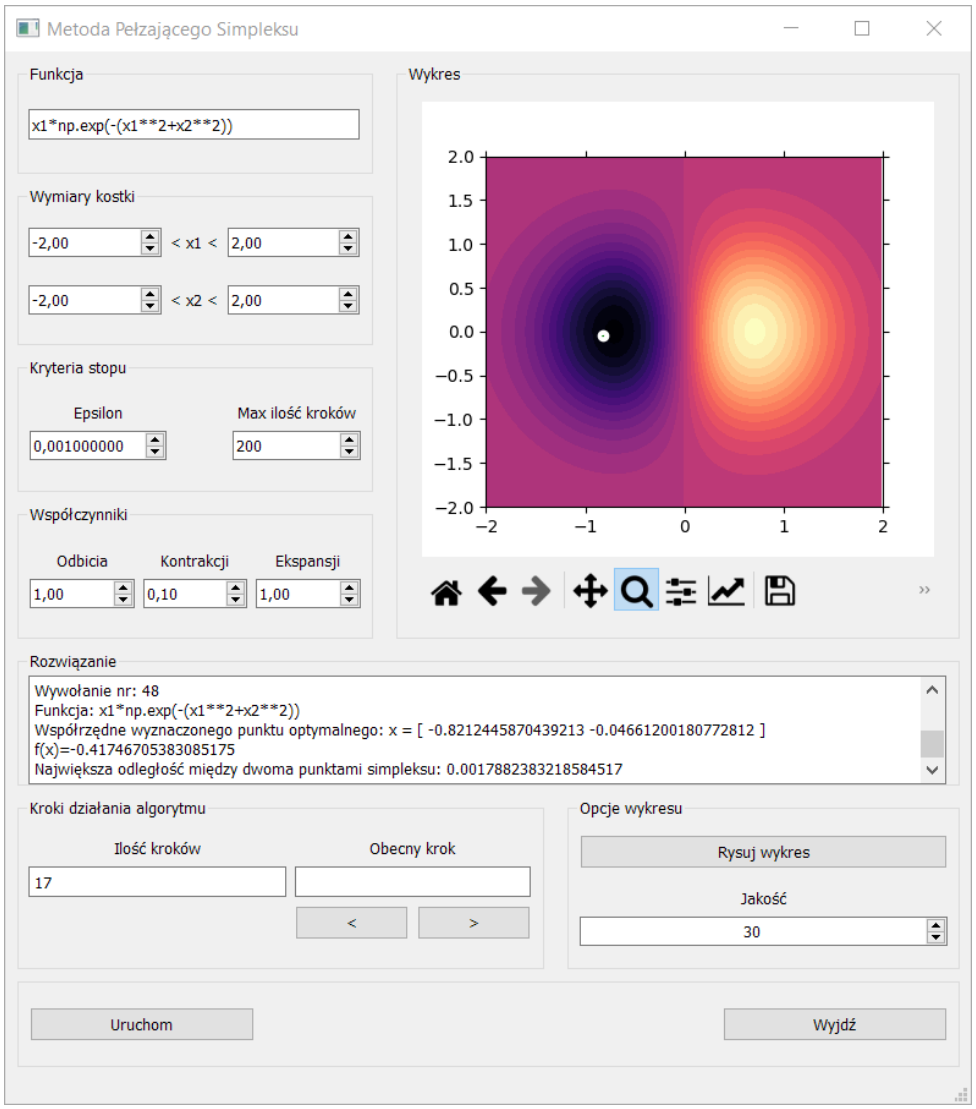


Rysunek 22: Wpływ za małej wartości współczynnika odbicia na otrzymany wynik

Współczynnik kontrakcji nie ma aż tak dużego wpływu na dokładność obliczeń jak współczynnik odbicia, ale ma większy niż współczynnik ekspansji. Zbyt duża wartość ponownie wydłuży ilość obliczeń a zbyt mała wartość doprowadzi do szybszego dojścia do minimum, ale z o wiele mniejszą dokładnością. Wyniki badań przedstawiono poniżej.



Rysunek 23: Wpływ za dużej wartości współczynnika kontrakcji na otrzymany wynik



Rysunek 24: Wpływ za małej wartości współczynnika kontrakcji na otrzymany wynik

Mając na uwadze powyższe badania dobrano wartości kryteriów i współczynników tak, aby wzajemnie się nie ograniczyły, funkcja wykonywała możliwie jak najmniej obliczeń a przy tym dokładność utrzymywała się na wysokim poziomie.
Do testów w kolejnych podrozdziałach zostały dobrane poniżesze wartości:

- $\alpha = 1.0$ - współczynnik odbicia,
- $\beta = 0.5$ - współczynnik kontrakcji,
- $\gamma = 1.0$ - współczynnik ekspansji,
- $\varepsilon = 0.001$ - dokładność obliczeń,
- $L = 200$ - liczba iteracji,

4.4 Funkcja Goldsteina-Price'a

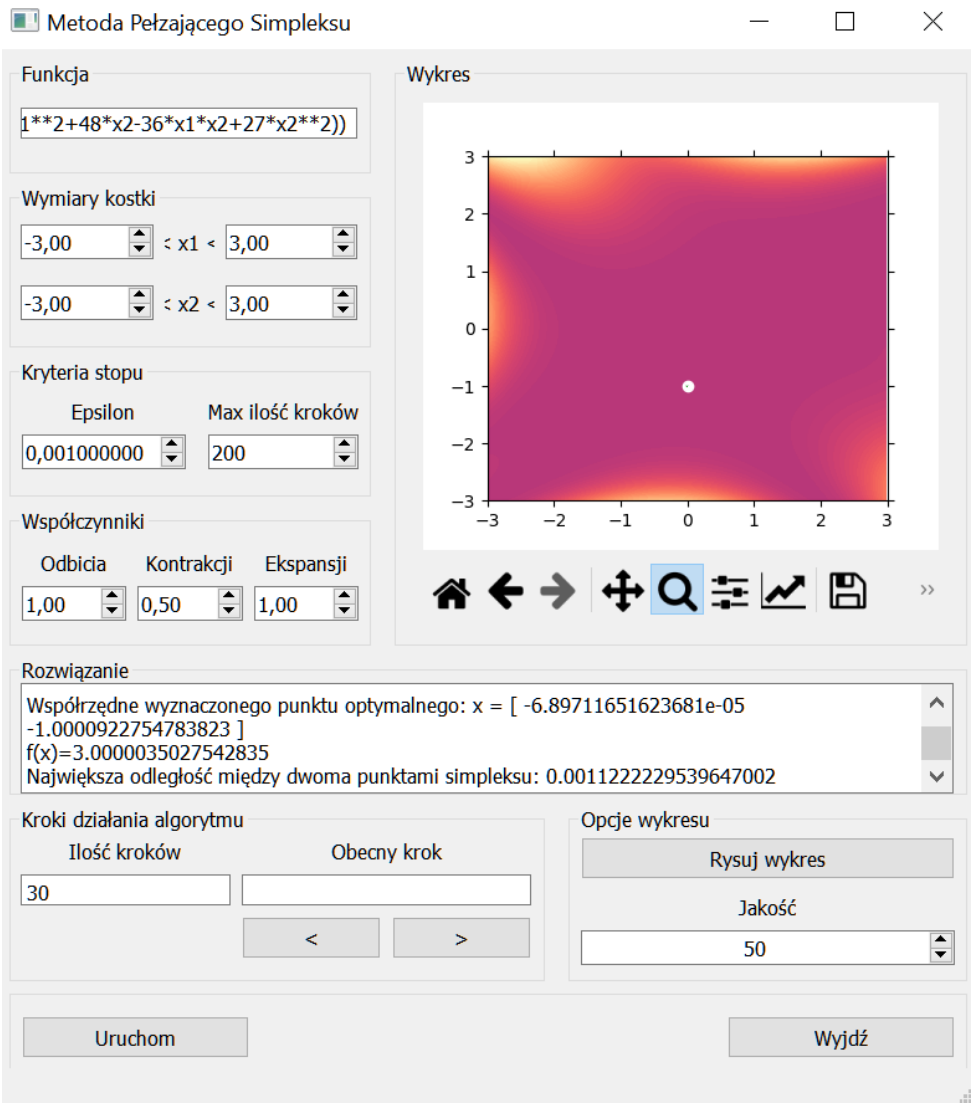
Funkcja Goldsteina-Price'a posiada trzy minimama lokalne i minimum globalne. Wygląda ona następująco:

$$f(x) = (1 + (x_1 + x_2 + 1)^2 \cdot (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x1x_2 + 3x_2^2)) \cdot (30 + (2x_1 - 3x_2)^2 \cdot (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2))$$

Wymiary kostki:

$$-3 \leqslant x_1 \leqslant 3$$
$$-3 \leqslant x_2 \leqslant 3$$

Minimum globalne $f(x^*) = 3$:

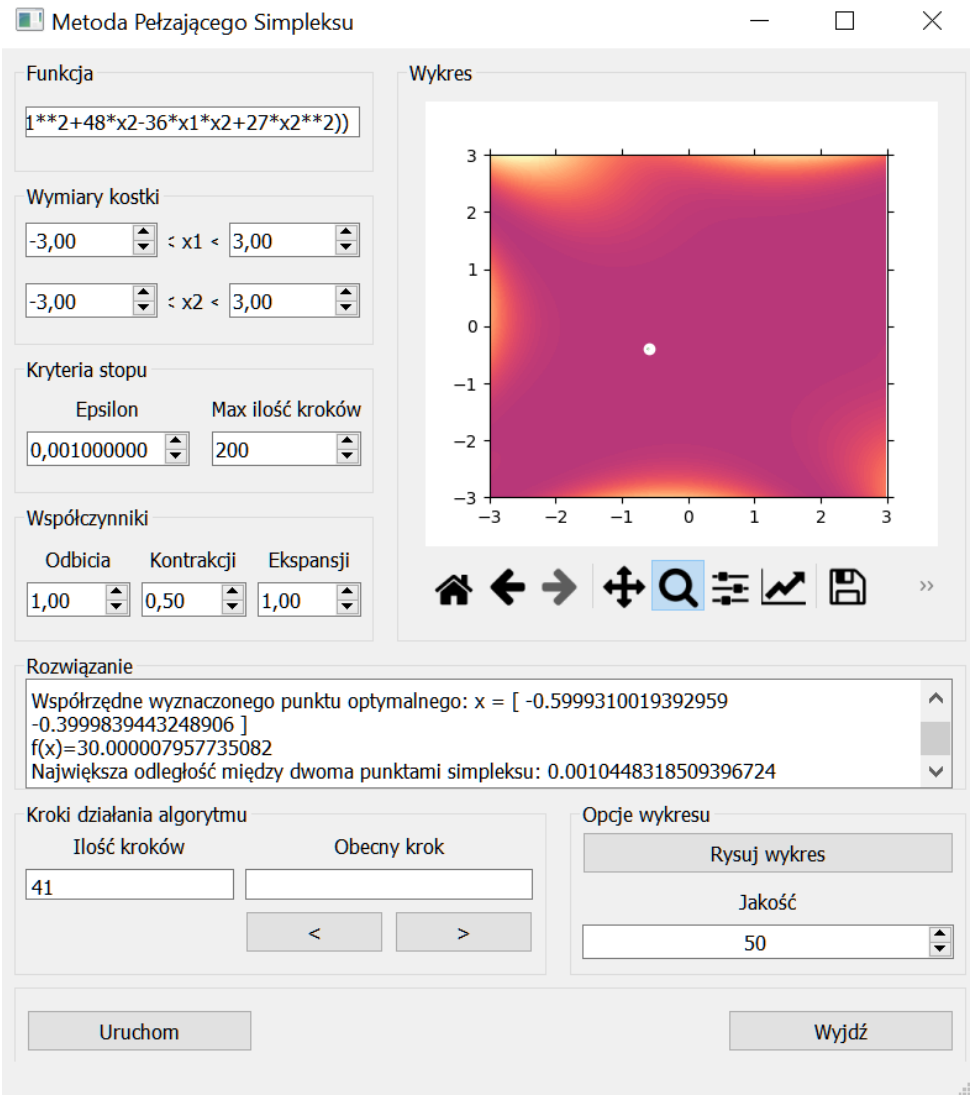


Rysunek 25: Minimum globalne funkcji Goldsteina-Price'a.

Współrzędne punktu, w którym funkcja osiąga minimum globalne:

$$x^* = [-6.89711 \cdot 10^{-5}, -1.000092]$$

Minimum lokalne $f(x^*) = 30$:

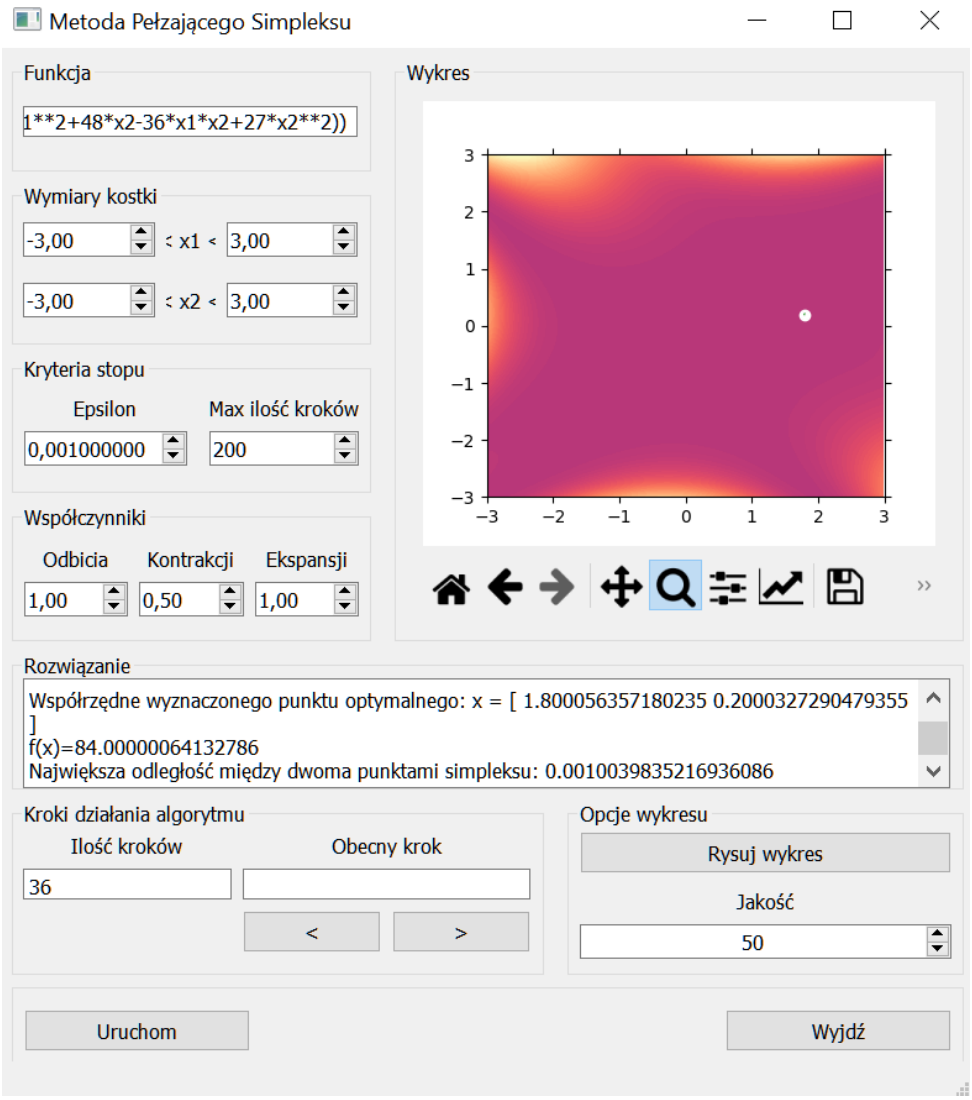


Rysunek 26: Pierwsze minimum lokalne funkcji Goldsteina-Price'a.

Współrzędne punktu, w którym funkcja osiąga pierwsze minimum lokalne:

$$x^* = [-0.599931, -0,399983]$$

Minimum lokalne $f(x^*) = 84$:

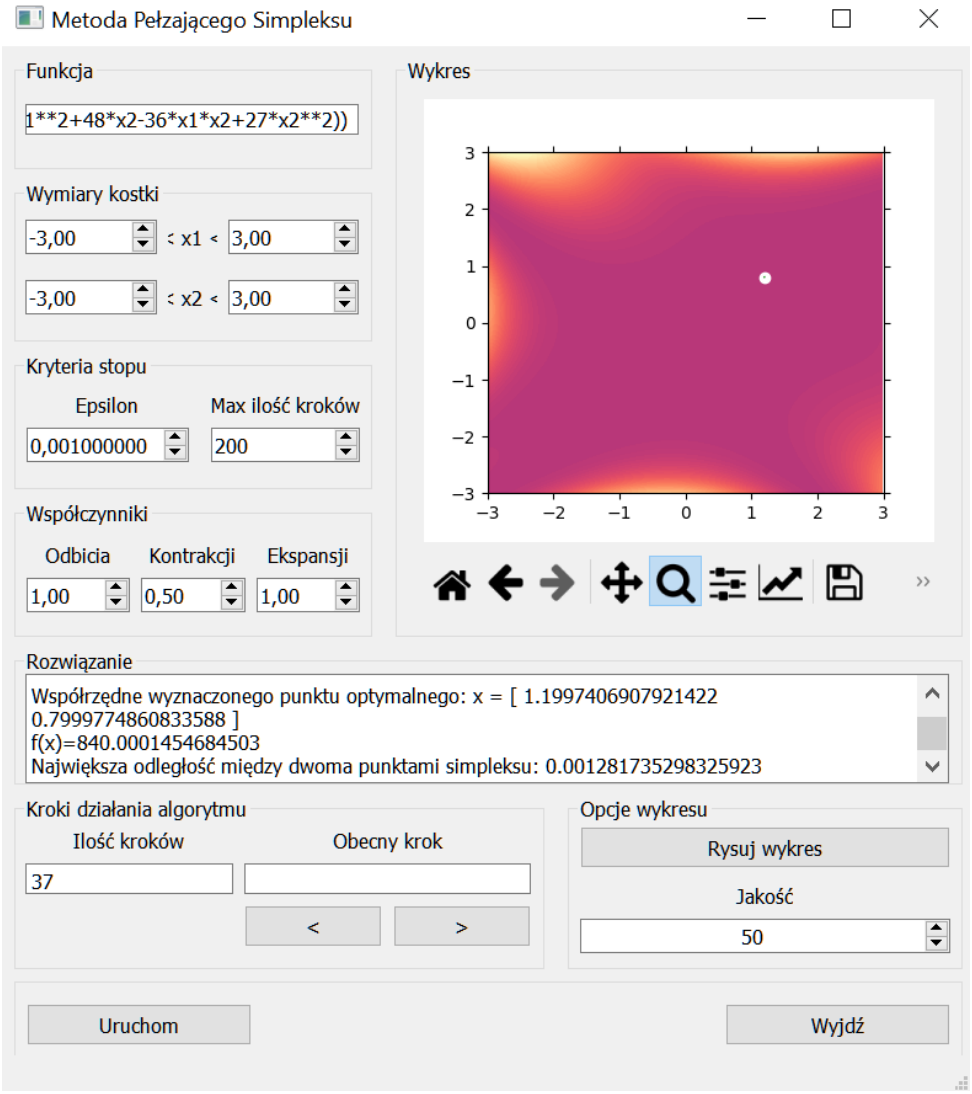


Rysunek 27: Drugie minimum lokalne funkcji Goldsteina-Price'a.

Współrzędne punktu, w którym funkcja osiąga drugie minimum lokalne:

$$x^* = [-1.800056, 0.200032]$$

Minimum lokalne $f(x^*) = 840$:



Rysunek 28: Trzecie minimum lokalne funkcji Goldsteina-Price'a.

Współrzędne punktu, w którym funkcja osiąga trzecie minimum lokalne:

$$x^* = [-1.199740, 0.799977]$$

Zaimplementowany przez nas algorytm poprawnie znalazł wszystkie wartości minimów lokalnych oraz minimum globalne. Najrzadziej wyliczał on wartość minimum równą 840. Na otrzymane minimum można również wpływać poprzez zmianę zakresu wartości kostki. Ograniczamy wtedy miejsce, w którym zostanie stworzony simpleks i dzięki temu wpłyniemy na jego zbieżność do konkretnego minimum.

4.5 Zmodyfikowana funkcja Himmelblau'a

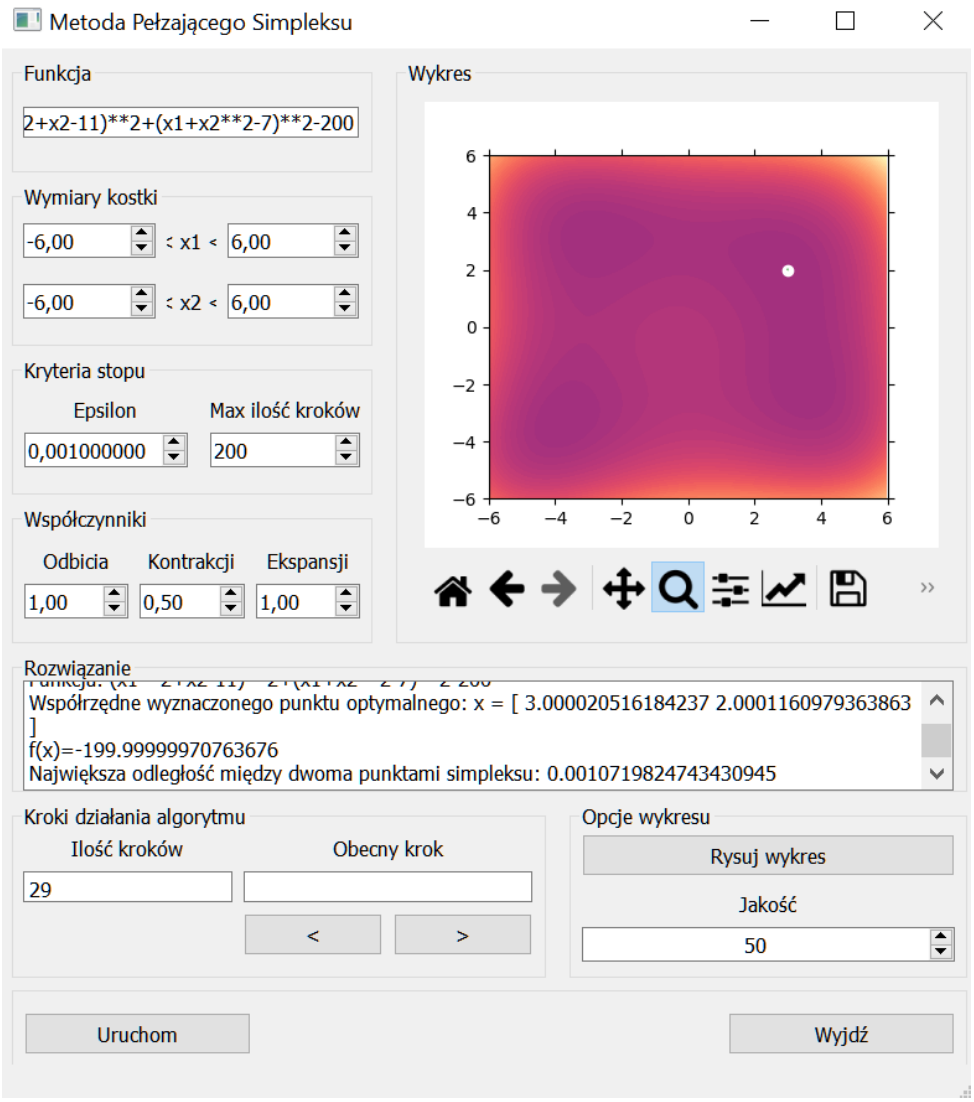
Zmodyfikowana funkcja Himmelblau'a posiada 4 minima lokalne o tej samej wartości $f(x^*) = -200$. Posiada ona następującą postać:

$$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 - 200$$

Wymiary kostki dla tej funkcji ustaliliśmy na poniższe wartości, aby znaleźć wszystkie minima lokalne:

$$-6 \leqslant x_1 \leqslant 6$$
$$-6 \leqslant x_2 \leqslant 6$$

Pierwsze minimum:



Rysunek 29: Pierwsze minimum lokalne funkcji Himmelblau'a.

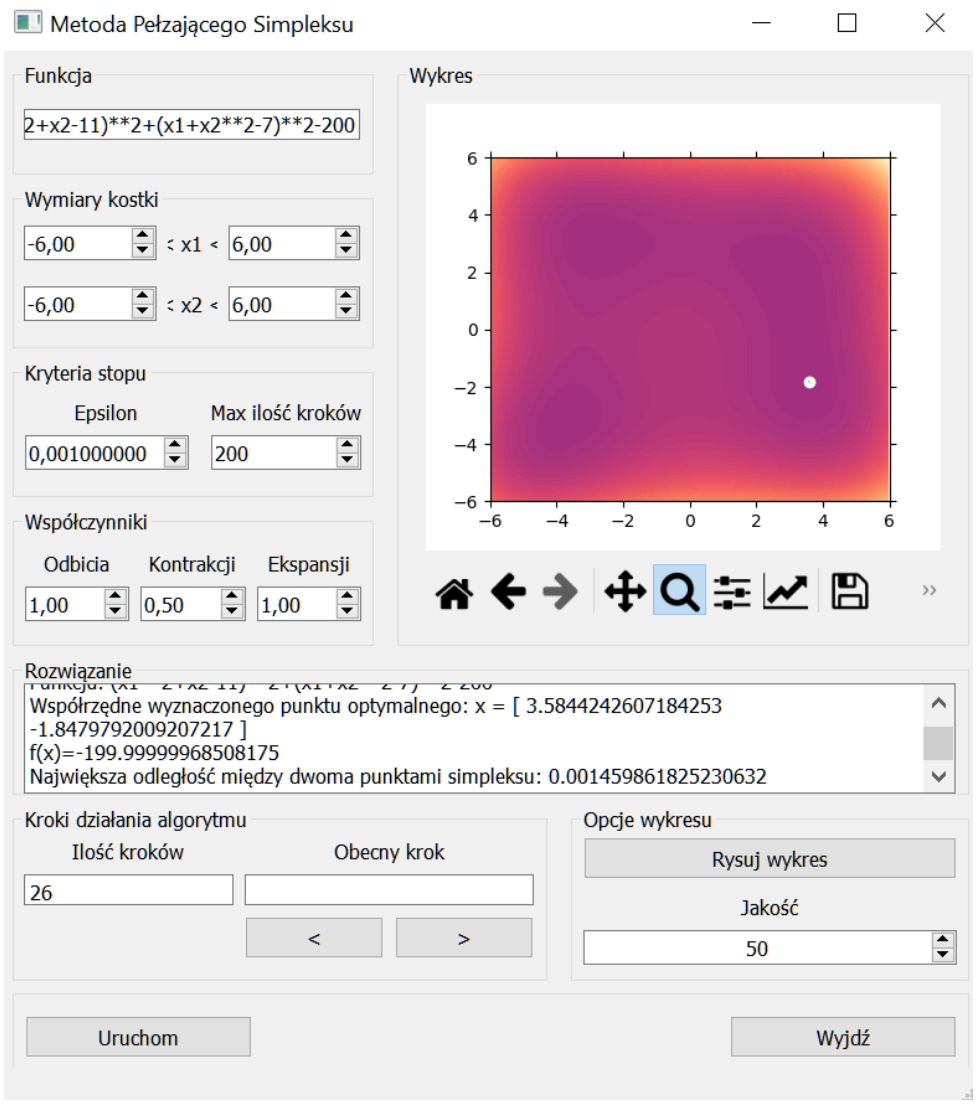
Współrzędne punktu, w którym funkcja osiąga pierwsze minimum lokalne:

$$x^* = [3.000020, 2.000116]$$

W tym punkcie funkcja osiąga wartość równą:

$$f(x^*) = -199,999999$$

Drugie minimum:



Rysunek 30: Drugie minimum lokalne funkcji Himmelblau'a.

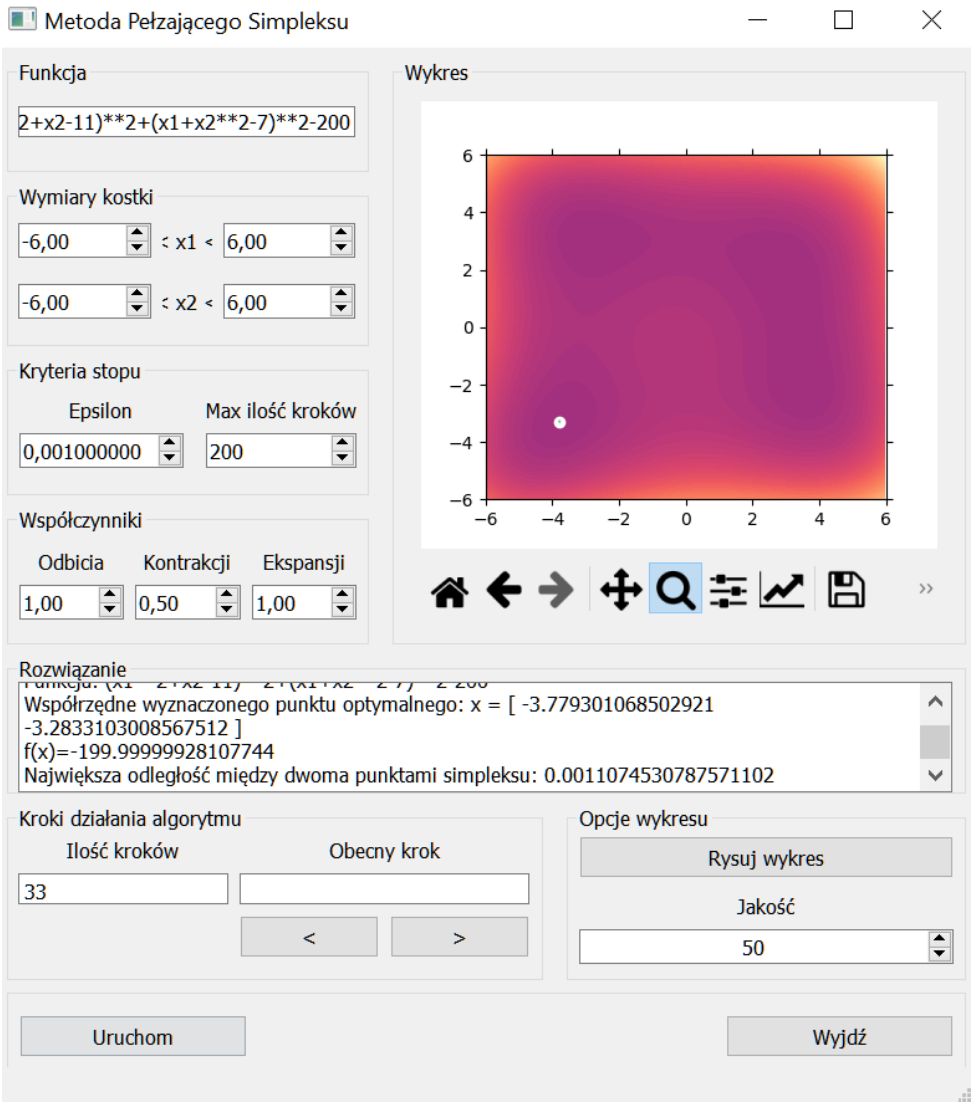
Współrzędne punktu, w którym funkcja osiąga drugie minimum lokalne:

$$x^* = [3.584424, -1.847979]$$

W tym punkcie funkcja osiąga wartość równą:

$$f(x^*) = -199,999999$$

Trzecie minimum:



Rysunek 31: Trzecie minimum lokalne funkcji Himmelblau'a.

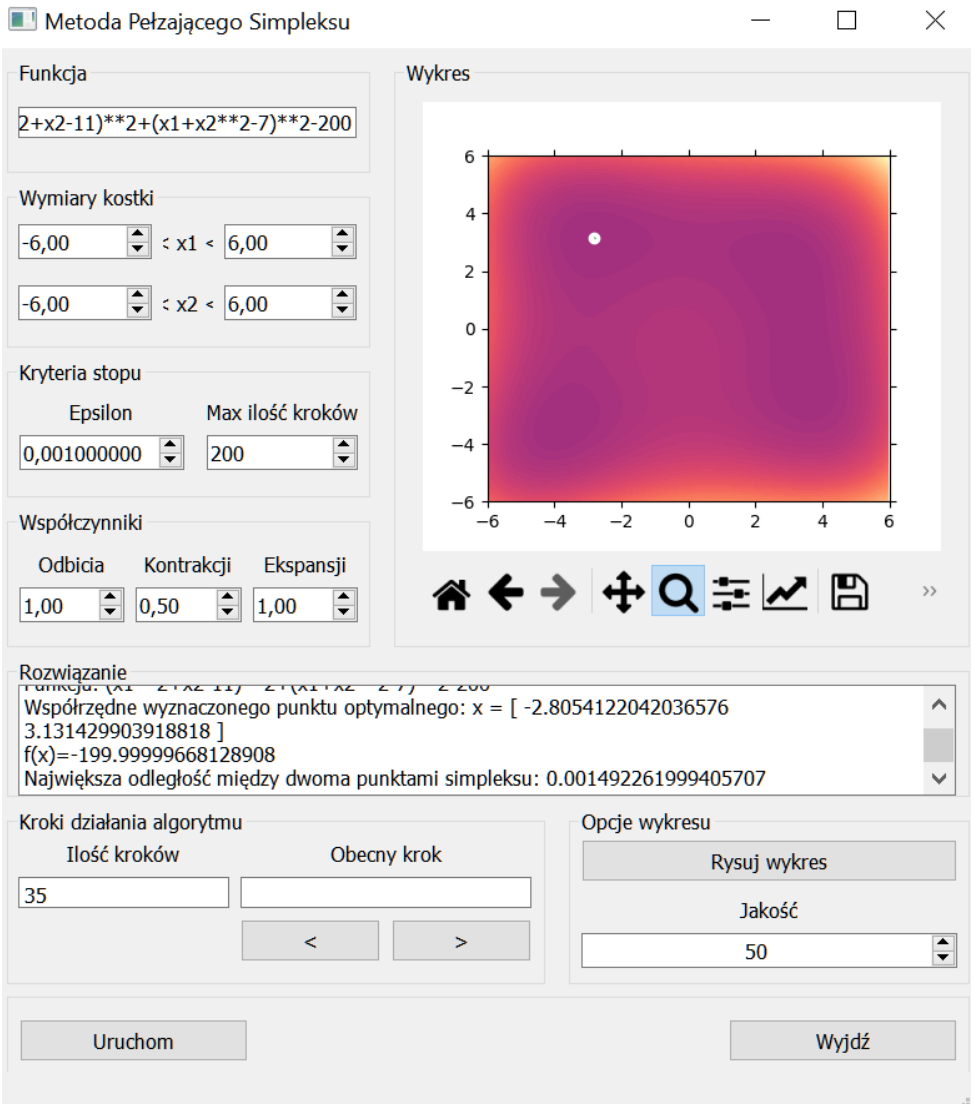
Współrzędne punktu, w którym funkcja osiąga trzecie minimum lokalne:

$$x^* = [-3.779301, -3.283310]$$

W tym punkcie funkcja osiąga wartość równą:

$$f(x^*) = -199,999999$$

Czwarte minimum:



Rysunek 32: Czwarte minimum lokalne funkcji Himmelblau'a.

Współrzędne punktu, w którym funkcja osiąga czwarte minimum lokalne:

$$x^* = [-2.805412, 3.131429]$$

W tym punkcie funkcja osiąga wartość równą:

$$f(x^*) = -199,999999$$

W podanym powyżej przypadku algorytm poprawnie znalazł cztery minima lokalne funkcji f(x). Podobnie jak w przypadku funkcji Goldsteina-Price'a zbieżność do danego punktu minimum mogliśmy wymuszać poprzez ograniczenie zakresu kostki z której losowane były wartości początkowe simpleksu.

4.6 Przykład z oddania programu

Dodatkowo podczas oddania części programowej zostaliśmy poproszeni o przetestowanie działania aplikacji i algorytmu dla dwóch poniższych funkcji:

$$f_1(x) = 0 - (2 \cdot x_1^3 + x_2^3 - 6 \cdot x_1 - 12 \cdot x_2),$$

$$f_2(x) = (2 \cdot x_1^3 + x_2^3 - 6 \cdot x_1 - 12 \cdot x_2)$$

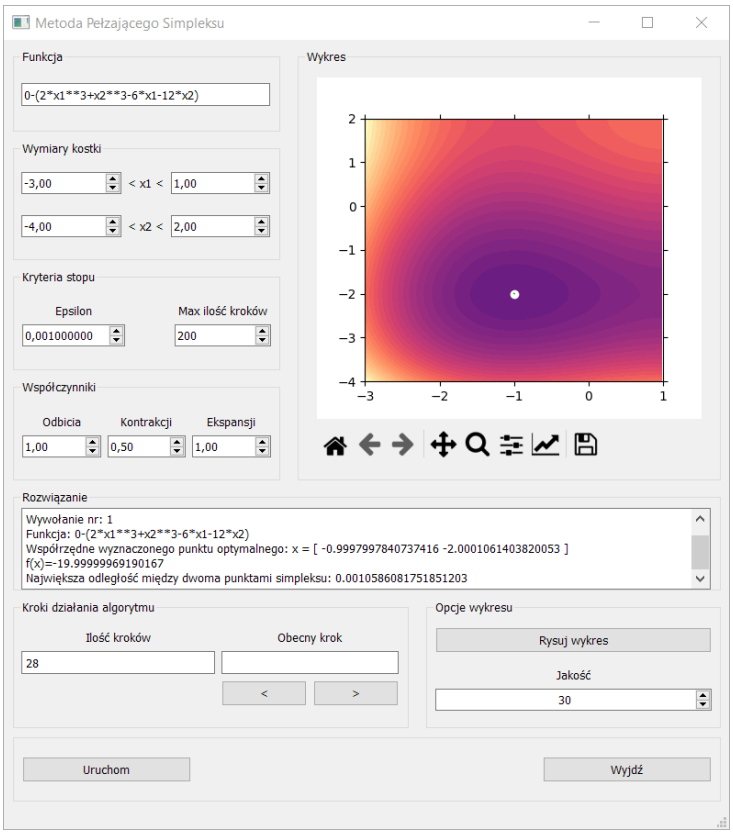
Wymiary kostki dla $f_1(x)$ ustaliliśmy na poniższe wartości po wykonaniu kilku testów:

$$-3 \leqslant x_1 \leqslant 1$$
$$-4 \leqslant x_2 \leqslant 2$$

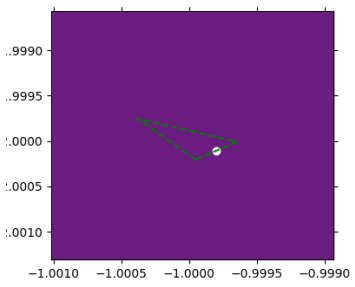
Otrzymane wyniki dla funkcji $f_1(x)$ przedstawiono poniżej:

$$x^* = [-0.9997997840737416 \ \ -2.00010614038200053],$$
$$f(x^*) = -19.99999969190167.$$

Funkcja zatrzymała się po wykonaniu 28 kroków, po osiągnięciu wartości granicznej ε dla największej odległości między dwoma punktami simpleksu równej 0.0010586081751851203.



(a) Wynik działania algorytmu dla funkcji $f_1(x)$



(b) Przybliżenie otrzymanej wartości minimum

Rysunek 33: Wynik działania algorytmu dla funkcji $f_1(x)$

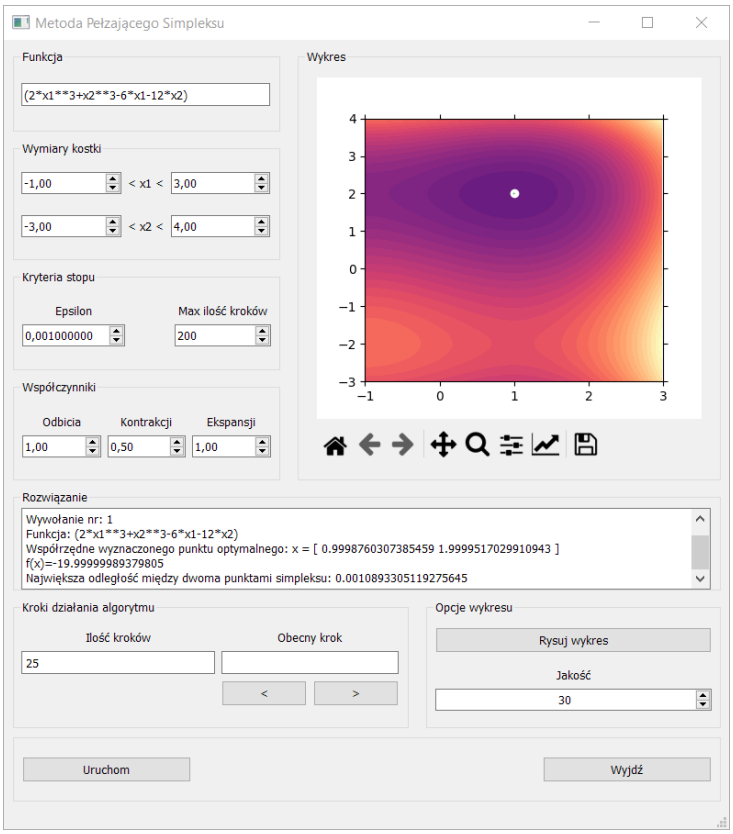
Wymiary kostki dla $f_2(x)$ ustaliliśmy na poniższe wartości po wykonaniu kilku testów:

$$-1 \leqslant x_1 \leqslant 3$$
$$-3 \leqslant x_2 \leqslant 4$$

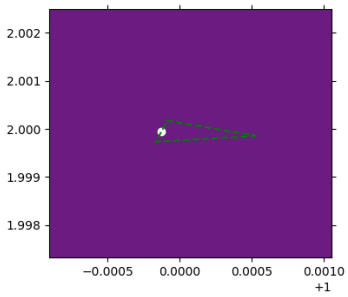
Otrzymane wyniki dla funkcji $f_2(x)$ przedstawiono poniżej:

$$x^* = [0.9998760307385459\ 1.9999517029910943],$$
$$f(x^*) = -19.99999989379805.$$

Funkcja zatrzymała się po wykonaniu 25 kroków, po osiągnięciu wartości granicznej ε dla największej odległości między dwoma punktami simpleksu równej 0.0010893305119275645.



(a) Wynik działania algorytmu dla funkcji $f_2(x)$



(b) Przybliżenie otrzymanej wartości minimum

Rysunek 34: Wynik działania algorytmu dla funkcji $f_2(x)$

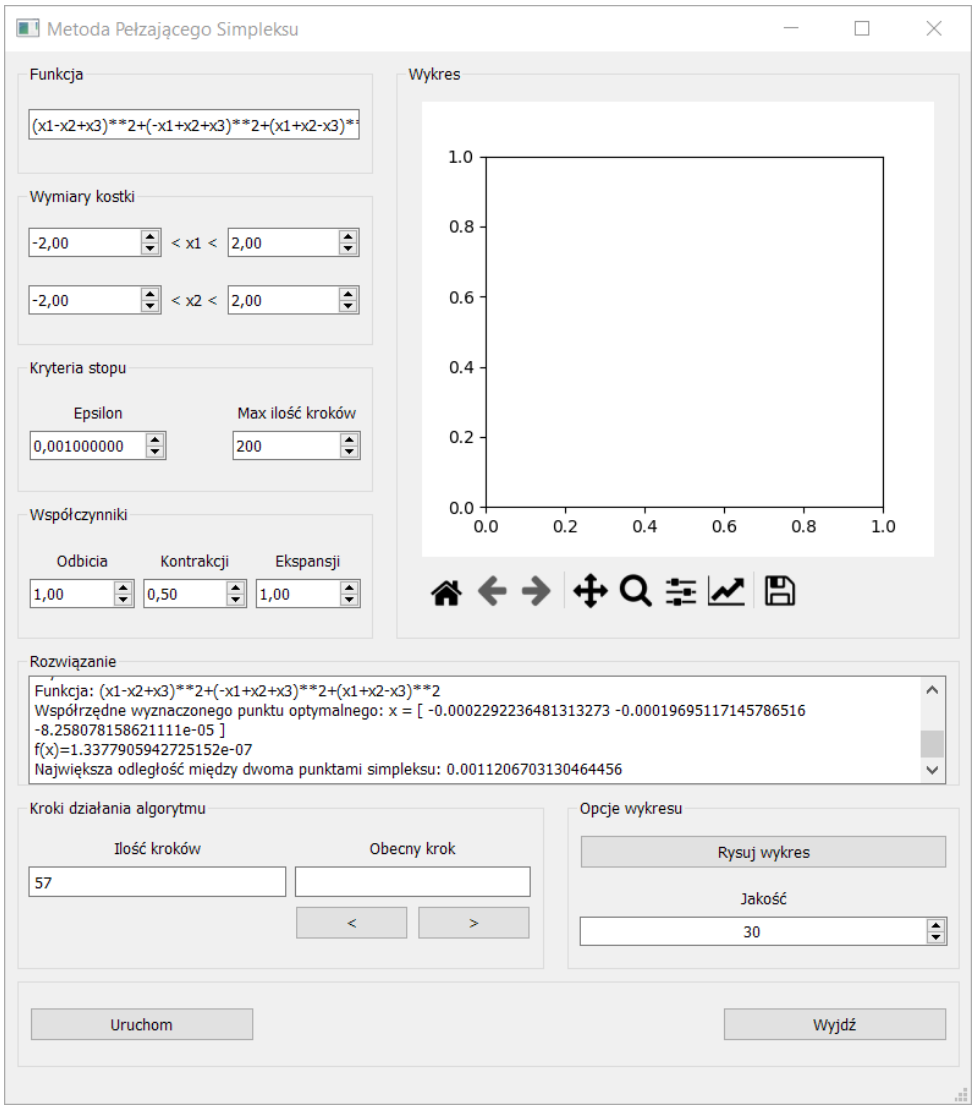
4.7 Funkcja wielu zmiennych

Do testów dla większej ilości zmiennych wybraliśmy funkcję Zangwilla. Posiada ona 3 wymiary i jest szczególnie trudnym przypadkiem dla metody Neldera-Meada.
Funkcja:

$$f(x) = (x_1 - x_2 + x_3)^2 + (-x_1 + x_2 + x_3)^2 + (x_1 + x_2 - x_3)^2$$

Wymiary kostki:

$$-2 \leqslant x_1 \leqslant 2$$
$$-2 \leqslant x_2 \leqslant 2$$



Rysunek 35: Wynik działania algorytmu dla funkcji Zangwilla

Po wywołaniu funkcji widzimy, że nie zostały wyrysowane warstwice. Jest to poprawne działanie programu. Aplikacja zakłada wizualizację tylko dla wymiarów zadania $n = 2$.

Oczekiwany wynik wynosi:

$$x^* = [0.0\ 0.0\ 0.0],$$
$$f(x) = 0.0\ .$$

Otrzymany wynik wynosi:

$$x^* = [-0.0002922236481313273\ -0.00019695117145786516\ -8.2580781586211111e-05],$$

$$f(x^*) = 1.3377905942725152e-07.$$

Jak widać otrzymana wartość jest bardzo zbliżona do oczekiwanej wartości. Algorytm wykonał obliczenia w 57 krokach i zatrzymał działanie ze względu na wartość największej odległości między dwoma punktami simpleksu, która osiągnęła granicę wyznaczaną przez kryterium stopu ε.

5 Napotkane błędy i problemy

Podczas implementacji oraz w fazie testowania algorytmu, natknęliśmy się na kilka problemów, które wymagały poprawy. Część z nich uniemożliwiała poprawne działanie metody a część z nich była jedynie zmianami estetycznymi. Niektóre problemy nie wymagały naprawy, były to jedynie błędy w rozumowaniu.

5.1 Punkt startowy

W implementacji nie ma możliwości wybrania samemu punktu startowego. Współrzędne simpleksu są losowane między granicami wymiarów kostki. Ma to znaczenie w przypadku funkcji wielomodalnych, w których znajduje się kilka minimów. Z racji tego osiągnięcie danego minimum będzie uzależnione od wylosowania wierzchołków startowych simpleksu. Na końcowy wynik można wpłynąć ograniczając odpowiednio wymiary kostki, co wpłynie na miejsce losowania punktów simpleksu. Nie jest to błąd w implementacji, a jedynie specyfika dobranego rozwiązania.

5.2 Dyskusja błędów obliczeń

Zaimplementowany algorytm jest algorytmem numerycznym, przez co uzyskane wyniki nie będą dokładnie takie same jak wartości oczekiwane. Będą natomiast bardzo dokładnymi przybliżeniami oczekiwanych wartości. Na dokładność otrzymywanego wyniku ma największy wpływ wartość kryterium stopu ε i pośrednio liczba iteracji dobrana tak, aby nie ograniczyć dojścia do wartości ε. Również wartości współczynników mają wpływ na osiągany wynik.

5.3 Składnia języka *Python* i biblioteki *numpy*

Przy wykorzystywaniu funkcji matematycznych w języku *Python* oraz tych z biblioteki *numpy* należy pamiętać o ich specyficznej składni. Przykładowo znak potęgowania nie wygląda tak: ^ a tak: ** a funkcja sinus musi zostać wpisana z przedrostkiem np "np.sin()" sugerując na wykorzystanie biblioteki *numpy*. Składnia ta może być niewygodna w użytkowaniu jednakże nie jest to błąd a specyfika implementacji kodu. Może on zostać rozwiązany przez odpowiednie importowanie pojedynczych funkcji jako konkretne nazwy (aby pozbyć się przedrostków "np.") oraz zaimplementować funkcje parsujące. My jednak zdecydowaliśmy się na pozostanie przy składni *Pythona* w celu łatwiejszej analizy innych możliwych błędów, co okazało się pomocne przy kolejnym problemie.

5.4 Przechowywanie danych

Podczas fazy testowej zauważono, że po kilkukrotnym wywołaniu algorytmu dla tej samej funkcji zaczynały się pojawiać błędne wyniki. Po głębszej analizie odkryto, że problemem był sposób magazynowania wszystkich odpowiedzi, dla każdego kroku. Lista ta nie była odpowiednio czyszczona i w momencie kolejnego wywołania programu łączyła otrzymane wyniki razem z wynikami poprzedniego wywołania. Po naprawieniu błędu, problem przestał istnieć i algorytm poprawnie wyliczał wartości.

5.5 Zabezpieczenia

Program nie pozwoli na uruchomienie algorytmu dla danej funkcji bez podania odpowiednich parametrów. Bazowo wpisane są wartości przedstawione na rys. 2 i to z nimi wywoła się funkcja, gdy nie zostaną one same wybrane. Dodatkowo na pola wyboru wartości zmiennych zostały nałożone ograniczenia dolne i górne, dzięki czemu można oscylować tylko w dozwolonych wartościach przez algorytm, np. współczynnik odbicia musi być dodatni ($\alpha > 0$) a współczynnik kontrakcji musi zawierać się między wartościami 0 i 1 ($0 < \beta < 1$).

6 Podsumowanie

- W ramach projektu udało się pomyślnie wykonać implementację metody pełzającego simpleksu Neldera-Meada oraz jej graficzną wizualizację.
- Metoda osiąga wyniki zgodne z testowymi funkcjami[1].
- Ze względu na specyfikę metody (algorytm numeryczny) osiągnięte rozwiązania nie będą w pełni zgodne z oczekiwanymi wartościami. Na dokładność uzyskanych wyników ma wpływ podana wartość ε.
- Funkcja pozwala na znalezienie wszystkich minimów funkcji wielomodalnych, jednakże ze względu na specyfikę wyboru punktu startowego (wybór losowej wartości z

zakresu wartości kostki), nie zawsze będziemy osiągać wybrane przez nas minimum. W takim wypadku należy odpowiednio ograniczyć wymiary kostki.

- Dla zwiększenia czytelności można zmienić dokładność liczb tak, aby nie miały za dużo miejsc po przecinku. Można także zaimplementować funkcje parsujące, które umożliwią wpisywanie wzorów bez specyficznej składni dla języka Python.
- Współczynnik oraz kryteria stopu mają znaczny wpływ na otrzymywane wyniki. Ich wartości do badań zostały dobrane po wykonaniu wcześniejszych badań.

Literatura

[1] Dr inż. Ewa Szlachcic. *Przykładowe zadania testowe dla problemów nieliniowych*. Zadania testowe dla problemów nieliniowych. Dostęp: 31.05.2021.

[2] Nalder John A. and Mead Roger. *A Simplex Method for Function Minimization*. A Simplex Method for Function Minimization. Dostęp: 31.05.2021.

[3] Beling Piotr and Wasilewski Filip. *Metody obliczeniowe optymalizacji*. Metoda pełzającego simpleksu Neldera-Meada. Dostęp: 31.05.2021.