OPTYMALIZACJA NIELINIOWA

PROJEKT NR 2

SEBASTIAN ZŁOTEK

Metoda gradientów sprzężonych

Gradient sprzężony (ang. conjugate gradient) to metoda optymalizacji niestosowana do zmiennych równoległych, która jest szybsza niż metoda gradientu prostego. Jest to metoda iteracyjna, która po każdym kroku aktualizuje kierunek poszukiwania minimum na podstawie poprzednich kierunków i gradientów. Dzięki temu możliwe jest uniknięcie oscylacji wokół minimum lokalnego, co pozwala na szybsze znalezienie minimum globalnego.

Metoda Polak-Riberie

Metoda gradientów sprzężonych Polak-Ribiere to wariant metody gradientów sprzężonych, który został opracowany przez Polaka i Ribiere'a w 1969 roku. Różni się on od standardowej metody gradientów sprzężonych tym, że wykorzystuje bardziej aktualny gradient w celu aktualizacji kierunku poszukiwania minimum. W szczególności, nowy kierunek jest określany jako suma aktualnego gradientu i skalarowej mieszanki poprzedniego kierunku i gradientu. Dzięki temu, metoda ta jest bardziej skuteczna w przypadku funkcji z dużymi równoległymi gradientami.

Metoda Polak-Riberie

Metoda gradientów sprzężonych Polak-Ribiere opiera się na iteracyjnym aktualizowaniu kierunku poszukiwania minimum na podstawie aktualnego gradientu i poprzednich kierunków. W szczególności, nowy kierunek jest określany jako suma aktualnego gradientu i skalarowej mieszanki poprzedniego kierunku i gradientu.

Konkretniej, w każdym kroku, aktualny gradient jest obliczany w punkcie aktualnej wartości zmiennej, a następnie kierunek poszukiwania zostaje zaktualizowany na podstawie aktualnego gradientu i poprzedniego kierunku. Skalarowa mieszanka jest obliczana jako stosunek iloczynu skalarnego między aktualnym gradientem a poprzednim gradientem i iloczyn skalarny między aktualnym gradientem a aktualnym kierunkiem.

Metoda Polak-Riberie

Warto zauważyć, że w odróżnieniu od metody gradientu prostego, która zawsze kieruje się w kierunku gradientu, metoda ta pozwala na wykorzystanie informacji z poprzednich kroków, dzięki czemu jest bardziej skuteczna w unikaniu minimum lokalnych i znajdowaniu minimum globalnego.

Metoda Polak-Riberie

W metodach gradientów sprzężonych kierunki poszukiwań wyznacza się według zasady:

$$\mathbf{d}^{(0)} = -\nabla f(\mathbf{x}^{(0)}),$$
$$\mathbf{d}^{(i+1)} = -\nabla f(\mathbf{x}^{(i+1)}) + \beta^{(i+1)}\mathbf{d}^{(i)}, \text{ gdzie}$$

- w *metodzie Polaka-Ribiere'a*:

$$\beta^{(i+1)} = \frac{\nabla f(\mathbf{x}^{(i+1)})^T \left(\nabla f(\mathbf{x}^{(i+1)}) - \nabla f(\mathbf{x}^{(i)})\right)}{\nabla f(\mathbf{x}^{(i)})^T \nabla f(\mathbf{x}^{(i)})};$$

Zalety metody Polak-Riberie

- Szybkość działania: metoda ta jest zazwyczaj szybsza niż metoda gradientu prostego, ponieważ wykorzystuje informację z poprzednich kroków, co pozwala na unikanie oscylacji wokół minimum lokalnego.
- Oszczędność obliczeniowa: metoda ta nie wymaga obliczania macierzy Hessianu, co oznacza mniejsze obciążenie obliczeniowe.
- Wysoka skuteczność: metoda ta jest bardziej skuteczna niż metoda gradientu prostego w znajdowaniu minimum globalnego, szczególnie w przypadku funkcji z dużymi równoległymi gradientami.

Zalety metody Polak-Riberie

- Prosta implementacja: metoda ta jest prosta do zaimplementowania i nie wymaga skomplikowanego kodu.
- Brak założeń o geometrii funkcji celu: metoda ta nie wymaga założenia o geometrii funkcji celu, takiej jak np. jej pochodna drugiego rzędu, co jest konieczne w przypadku metody Newtona.

Wady metody Polak-Riberie

- Sensytywność na złe dobranie początkowego kierunku: metoda ta jest czuła na dobranie początkowego kierunku, co może prowadzić do zbyt wolnego lub nieconvergentnego działania.
- Wymaga dokładnego wyboru parametru: metoda ta wymaga dokładnego wyboru parametru aktualizacji kierunku, co może być trudne w niektórych przypadkach.
- Może zatrzymać się w minimum lokalnym: metoda ta może zatrzymać się w minimum lokalnym jeśli gradient jest zbyt mały, aby kontynuować poszukiwanie.

Wady metody Polak-Riberie

- Może być zwolniona przy dużych rozmiarach problemu: metoda ta może być zwolniona przy dużych rozmiarach problemu, ponieważ wymaga przechowywania wszystkich poprzednich kierunków.

- Może być trudna do zastosowania w problemach z nieograniczeniami: metoda ta jest oparta na gradientach i może być trudna do zastosowania w problemach z nieograniczeniami.

Użyta funkcja – Funkcja Levy’ego nr 13

- Funkcja Levy'ego nr 13 jest jedną z testowych funkcji optymalizacji, która jest często używana do testowania algorytmów optymalizacji. Jest to funkcja dwuwymiarowa, której wzór jest następujący:

$$f(x,y) = \sin^2(3\pi x) + (x-1)^2(1+\sin^2(3\pi y)) + (y-1)^2(1+\sin^2(2\pi y))$$

- Funkcja ta posiada jedno globalne minimum w punkcie (1,1) o wartości 0, jednak posiada także wiele minimum lokalnych i saddle pointów co czyni ją trudną do znalezienia globalnego minimum. Jest to funkcja o skomplikowanej geometrii, która jest często używana do testowania algorytmów optymalizacji, które są odporne na minimum lokalne i trudności związane z geometrią funkcji.

Opis kodu algorytmu Ternary

- Kod jest implementacją algorytmu ternary search, który jest metodą optymalizacji numerycznej. Algorytm ta przyjmuje jako argumenty funkcję f, dolne i górne granice przedziału poszukiwań oraz tolerancję. W pętli while, algorytm dzieli przedział poszukiwań na trzy części i porównuje wartości funkcji w dwóch punktach pośrodku przedziału. W zależności od tego, która z tych wartości jest mniejsza, algorytm dostosowuje dolne lub górne granice przedziału poszukiwań. Pętla jest kontynuowana, dopóki różnica między górną a dolną granicą przedziału poszukiwań jest większa niż podana tolerancja. Na końcu funkcja zwraca wartość środka przedziału poszukiwań.

Kod algorytmu Ternary

```r
# Algorytm Ternary
ternary <- function(f, lower, upper, tol) {
  f.lower <- f(lower)
  f.upper <- f(upper)
  while (abs(upper - lower) > 2 * tol) {
    x1 <- (2 * lower + upper) / 3
    f.x1 <- f(x1)
    x2 <- (lower + 2 * upper) / 3
    f.x2 <- f(x2)
    if (f.x1 < f.x2) {
      upper <- x2
      f.upper <- f.x2
    } else {
      lower <- x1
      f.lower <- f.x1
    }
  }
  return((upper + lower) / 2)
}
```

Opis kodu algorytmu Polak-Riberie

Kod jest implementacją algorytmu Polaka-Riberiego, który jest metodą optymalizacji numerycznej stosowaną do znajdowania minimum lokalnego funkcji nieograniczonej. Algorytm przyjmuje jako argumenty funkcję f, punkt startowy x oraz tolerancję.

Wewnątrz funkcji jest zadeklarowana zmienna beta, która jest ustawiona na 1. Zmienna d jest ustawiona jako wektor gradientu funkcji f z punktu x przemnożony przez -1.

Następnie zmienna i jest ustawiona na 0. W pętli repeat algorytm wykorzystuje funkcję ternary do znalezienia kroku, który jest wykorzystywany do przesunięcia punktu x w kierunku gradientu funkcji. Nowy punkt jest obliczany jako punkt x plus krok razy wektor gradientu.

Jeśli odległość między nowym punktem a punktem x jest mniejsza niż tolerancja, algorytm kończy działanie i zwraca nowy punkt. W przeciwnym razie, algorytm aktualizuje wartość beta, wektora d i punktu x, a także zwiększa licznik iteracji.

Funkcja wypisuje wartość wektora d, kroku, bety, liczby iteracji oraz punktu x każdą iterację.

Kod algorytmu Polak-Riberie

```
library(numDeriv) # Biblioteka do funkcji grad
```

```
# Algorytm Polaka-Riberie'go
polak_riberie <- function(f, x, tol) {
  beta <- 1
  d <- -grad(f,x)
  i <- 0
  repeat {
    g <- function(a) {f(x + a * d)}
    step <- ternary(g,0,5,tol)
    new.x <- x + step * d
    if (dist(rbind(new.x,x)) < tol) {
      cat("Wektor: ", d, "\n Krok: ", step,
          "\n Beta: ", beta,"\n Iteracja: ", i, "\n Punkt: ", new.x,"\n\n")
      return(new.x)
    }
    beta <- grad(f,new.x) %*% ((grad(f,new.x))-(grad(f,x))) / (grad(f,x) %*% grad(f,x))
            d <- -grad(f,new.x) + as.vector(beta) * d
            x <- new.x
            cat("Wektor: ", d, "\n Krok: ", step,
                "\n Beta: ", beta,"\n Iteracja: ", i, "\n Punkt: ", new.x,"\n\n")
            i <- i + 1
  }
}
```

Wizualizacja w języku R

Do wizualizacji w R studio został użyty pakiet rgl, mający na celu tworzenie wykresów w trybie 3D

```
fn1 <- function(x, y) {
  return(sin(3 * pi * x)^2 + (x - 1)^2 * (1 + sin(3 * pi * y)^2) + (y - 1)^2 * (1 + sin(2 * pi * y)^2))
} # Funkcja na dwoch zmiennych x,y potrzebna do dzialania wykresu
```

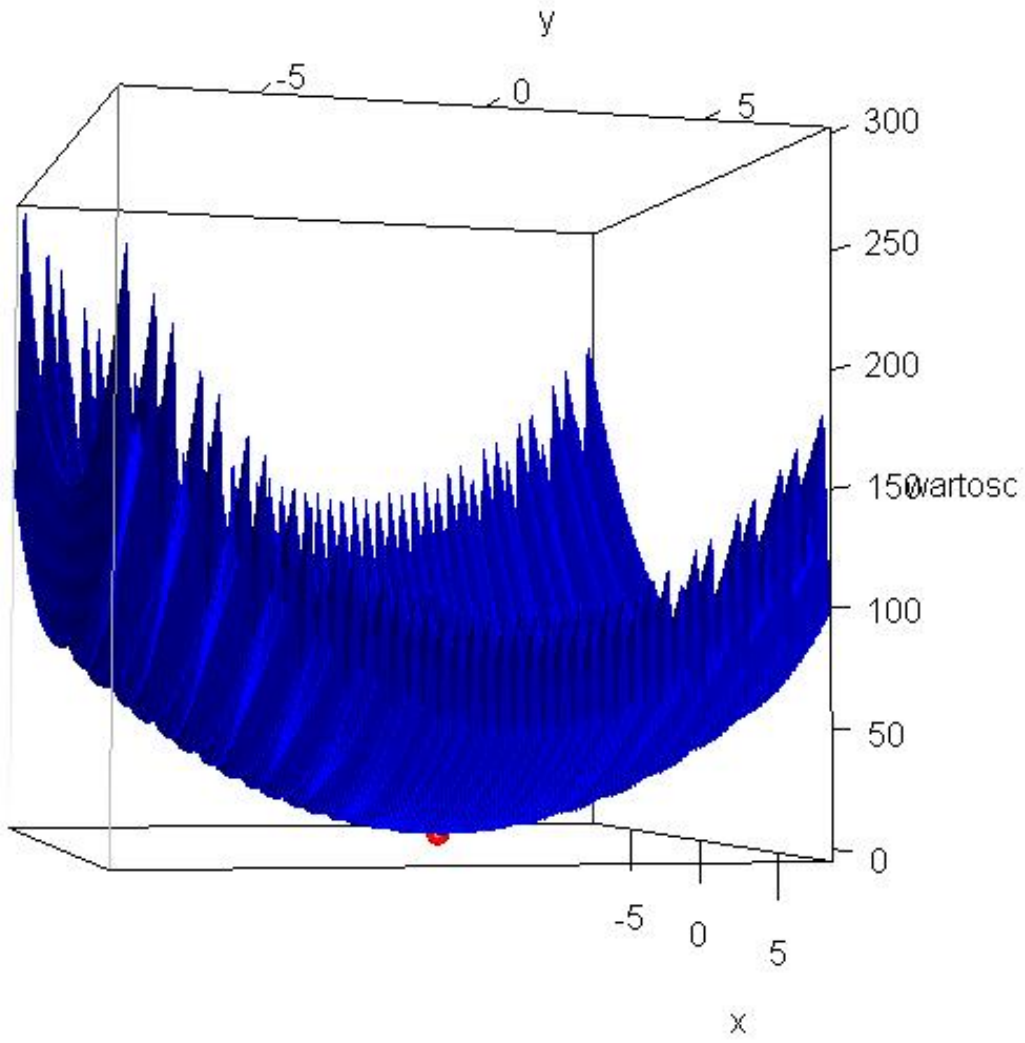
```
library(rgl) # Biblioteka do wykresow
```

Na zrzutach ekranu powyżej widać implementacje funkcji Levy'ego nr 13 w R oraz użyty pakiet rgl.

Wizualizacja w - języku R, pełen wykres funkcji

```
### Przechodze do wizualizacji ###
x <- seq(-8, 8, by = 0.01) # Wybieram zakres x w jakim ma dzialac funkcja
y <- seq(-8, 8, by = 0.01) # Wybieram zakres y w jakim ma dzialac funkcja
wartosc <- outer(x, y, fn1) # Wartosci funkcji
persp3d(x, y, wartosc, col = "blue") # Wpisuje dane na wykres
rgl.light(x=5, y=5, z=5, ambient="white", diffuse="white") # Rozjasniam wykres
minimum_funkcji <- c(1,1,0) # Wpisuje do wektora minimum funkcji
rgl.spheres(minimum_funkcji, radius=3, color="red") # Zaznaczam minimum na wykresie
```

Wizualizacja w - języku R, pełen wykres funkcji

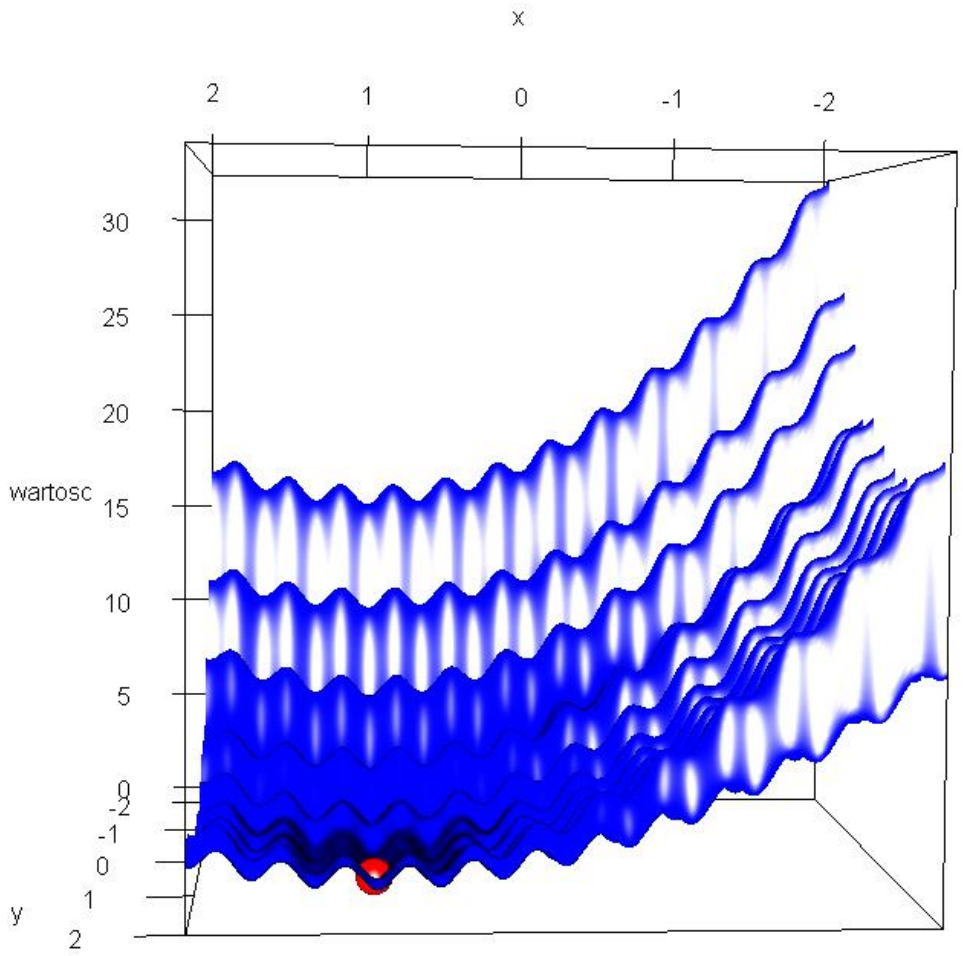


Jak widać na wykresie minimum znajduje się niemalże na środku (czerwona kropka)

Wizualizacja z użyciem mniejszego zakresu

```
### Podaje mniejszy zakres ###
x <- seq(-2, 2, by = 0.01) # Wybieram mniejszy zakres x w jakim ma dzialac funkcja
y <- seq(-2, 2, by = 0.01) # Wybieram mniejszy zakres y w jakim ma dzialac funkcja
wartosc <- outer(x, y, fn1) # Wartosci funkcji
persp3d(x, y, wartosc, col = "blue") # Wpisuje dane na wykres

rgl.light(x=2, y=2, z=2, ambient="white", diffuse="white") # Rozjasniam wykres
minimum_funkcji <- c(1,1,0) # Wpisuje do wektora minimum funkcji
rgl.spheres(minimum_funkcji, radius=0.5, color="red") # Zaznaczam minimum na wykresie
```

Wizualizacja z użyciem mniejszego zakresu



Na wykresie niemalże widać dokładne współrzędne minimum globalnego funkcji (czerwona kropka)

Działanie algorytmu

Aby algorytm działał poprawnie w miejsce x wstawiłem x[1], a y x[2]:

```
fn2 <- function(x) {
  return(sin(3 * pi * x[1])^2 + (x[1] - 1)^2 * (1 + sin(3 * pi * x[2])^2) + (x[2] - 1)^2 * (1 + sin(2 * pi * x[2])^2))
} # Funkcja na dwoch zmiennych x[1],x[2] potrzebna do dzialania algorytmu
```

Następnie wybrałem tolerancję, punkt początkowy oraz uruchomiłem algorytm z wcześniej wymienionymi wyborami:

```
tol <- 1e-15 # Ustalam tolerancje

punkt <- c(10201,10201) # Ustalam punkt początkowy

polak_riberie(fn2, punkt, tol) # Wczytuje algorytm z danymi
```

Działanie algorytmu dla punktu (10201, 10201)

- Dla wymienionych w tytule współrzędnych (10201, 10201) już ostatecznie możemy policzyć minimum, wpisując liczby współrzędnych większe od podanych program nie działa już poprawnie. Policzenie minimum w tym przypadku zajmuje 13 iteracji.

Działanie algorytmu dla punktu (10201, 10201)

```
> tol <- 1e-15 # Ustalam tolerancje
> punkt <- c(10201,10201) # Ustalam punkt początkowy
> polak_riberie(fn2, punkt, tol) # Wczytuje algorytm z danymi
Wektor:  -3031.096 1896.355
 Krok:  0.4345055
 Beta:  0.01115445
 Iteracja:  0
 Punkt:  1337.087 -1071.356

Wektor:  69.20635 95.05619
 Krok:  0.4521074
 Beta:  -0.002307661
 Iteracja:  1
 Punkt:  -33.29391 -213.9999

Wektor:  69.38426 484.5873
 Krok:  1.805871
 Beta:  3.679125
 Iteracja:  2
 Punkt:  91.68384 -42.34069

Wektor:  -189.2885 0.58134
 Krok:  0.08943831
 Beta:  -0.05362452
 Iteracja:  3
 Punkt:  97.88945 0.9999841

Wektor:  -0.1218309 -0.8253102
 Krok:  0.5118612
 Beta:  0.0006302291
 Iteracja:  4
 Punkt:  1.000014 1.297549
```

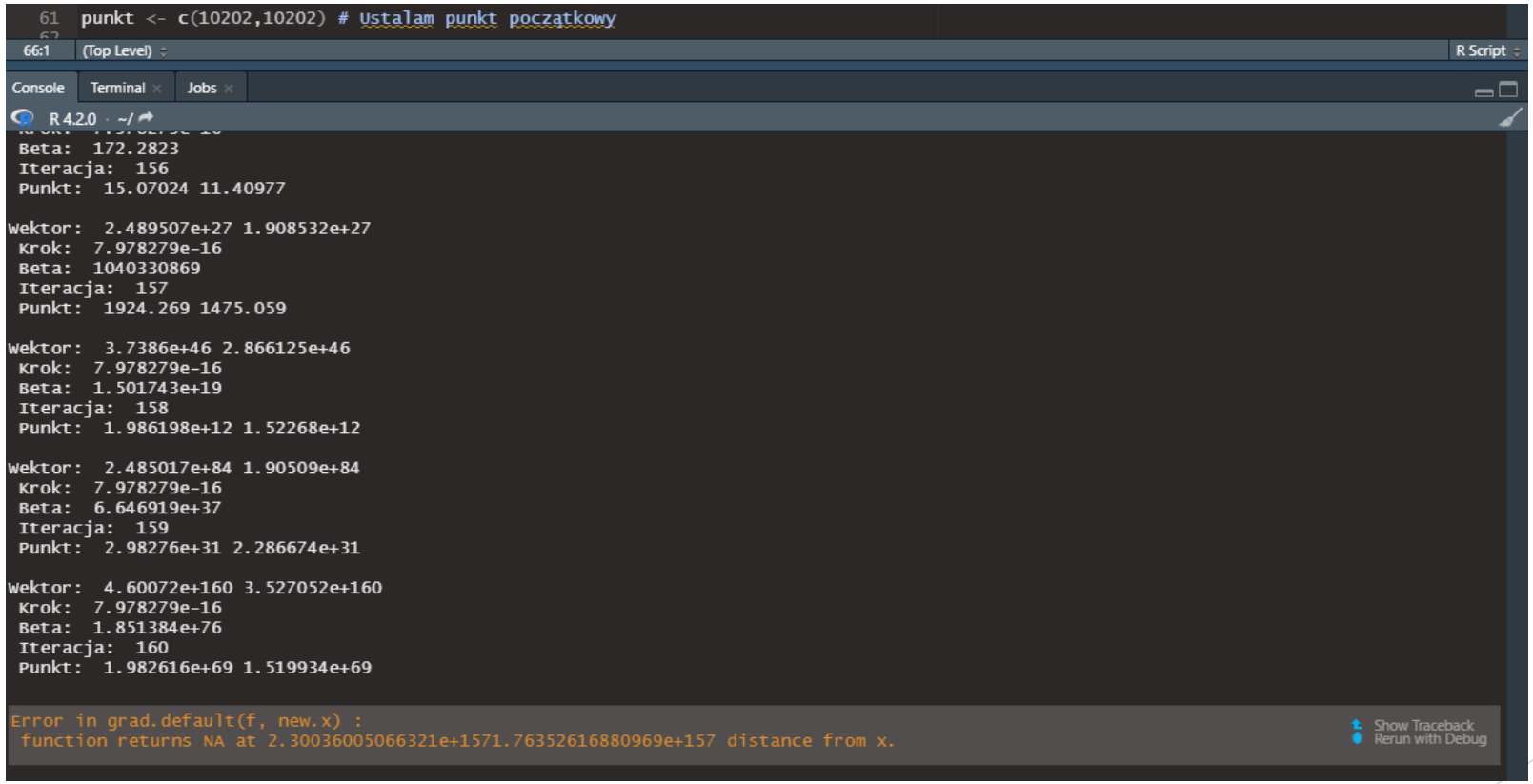
```
Wektor:  1.161533 -19.85369
 Krok:  0.1867162
 Beta:  23.33967
 Iteracja:  5
 Punkt:  0.9772663 1.143451

Wektor:  1.763099 4.872379
 Krok:  0.009737986
 Beta:  -0.2393995
 Iteracja:  6
 Punkt:  0.9885773 0.9501158

Wektor:  -0.01066367 0.2681475
 Krok:  0.006779331
 Beta:  0.04796209
 Iteracja:  7
 Punkt:  1.00053 0.9831473

Wektor:  0.01940681 0.06778628
 Krok:  0.06121245
 Beta:  0.249522
 Iteracja:  8
 Punkt:  0.9998772 0.9995612

Wektor:  -7.745315e-06 0.000199181
 Krok:  0.006346645
 Beta:  0.002685536
 Iteracja:  9
 Punkt:  1 0.9999914
```

```
Wektor:  1.059403e-09 3.700484e-09
 Krok:  0.04302234
 Beta:  1.834345e-05
 Iteracja:  10
 Punkt:  1 1

Wektor:  -3.545999e-14 9.13447e-13
 Krok:  0.006314336
 Beta:  0.000223787
 Iteracja:  11
 Punkt:  1 1

Wektor:  -1.29724e-15 -4.586675e-15
 Krok:  0.0453983
 Beta:  -0.007519649
 Iteracja:  12
 Punkt:  1 1

Wektor:  -1.29724e-15 -4.586675e-15
 Krok:  0.0363081
 Beta:  -0.007519649
 Iteracja:  13
 Punkt:  1 1

[1] 1 1
```

Działanie algorytmu dla punktu (10202, 10202) – nieco większego

- Algorytm w tym wypadku nie jest w stanie obliczyć minimum, program zwraca wartości NA, co uniemożliwia znalezienie najmniejszej wartości.

Działanie algorytmu dla punktu (10202, 10202) – nieco większego



Działanie algorytmu dla punktu (10000, 10000) – mniejszego

- Dla tego punktu w 46 iteracjach jesteśmy w stanie policzyć minimum

Działanie algorytmu dla punktu (10000, 10000) – mniejszego

```
punkt <- c(10000,10000) # Ustalam punkt początkowy
```

66:1 (Top Level) R Scrip

Console Terminal Jobs

R 4.2.0 · ~/

```
 Iteracja:  40
 Punkt:  1.000617 1.010829

Wektor:  0.004987072 -0.02729832
 Krok:  0.02389533
 Beta:  0.06047321
 Iteracja:  41
 Punkt:  0.999963 1.000204

Wektor:  -4.017514e-06 -6.592792e-05
 Krok:  0.007429679
 Beta:  0.002311077
 Iteracja:  42
 Punkt:  1 1.000001

Wektor:  -3.373647e-10 1.846606e-09
 Krok:  0.02153433
 Beta:  -2.759568e-05
 Iteracja:  43
 Punkt:  1 1

Wektor:  -4.373221e-14 -7.166115e-13
 Krok:  0.007393065
 Beta:  -0.0003750528
 Iteracja:  44
 Punkt:  1 1

Wektor:  -1.226098e-15 6.929082e-15
 Krok:  0.01789404
 Beta:  -0.007724041
 Iteracja:  45
 Punkt:  1 1

Wektor:  -1.226098e-15 6.929082e-15
 Krok:  0.1281813
 Beta:  -0.007724041
 Iteracja:  46
 Punkt:  1 1

[1] 1 1
```

Wnioski

Można powiedzieć, że algorytm Polaka-Riberiego jest skutecznym narzędziem do znajdowania minimum lokalnego funkcji nieograniczonej. W połączeniu z algorytmem ternary search, który jest używany do znajdowania kroku, pozwala on na efektywne przesuwanie punktu w kierunku minimum.

Funkcja Levy 13 jest testową funkcją optymalizacyjną, która jest złożona i posiada wiele lokalnych minimum, co daje dobry test dla algorytmu Polaka-Riberiego.

Wszystko to z trójwymiarowym wykresem naprawdę daje przejrzysty efekt co pozwala na łatwe określenie minimum.

Źródła

- https://www.sfu.ca/~ssurjano/levy13.html
- https://journalofinequalitiesandapplications.springeropen.com/articles/10.1186/s13660-021-02554-6
- https://link.springer.com/content/pdf/10.1186/s13660-021-02554-6.pdf