131890-S Metody Optymalizacji **Letni 2023/2024**

Zajęcia 4: Metody lokalne

Daniel Kaszyński

4.1 Pochodna kierunkowa i pochodna cząstkowa

Pochodna kierunkowa funkcji mierzy, jak funkcja zmienia się w określonym punkcie w kierunku danego wektora. Innymi słowy, określa szybkość zmian funkcji w określonym kierunku. Mierzy on wpływ przesunięcia funkcji f o wektor h.

Definicja 1: Pochodna funkcji jednowymiarowej

Pochodną funkcji $f : \mathbb{R} \to \mathbb{R}$ nazwiemy funkcję:

$$f'(x) = \frac{df}{dx}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{4.1}$$

Kiedy mamy do czynienia z funkcją jednowymiarową, domyślnie zakładamy, że wektor h jest po prostu równy 1 (patrz Definicja 1). Oznacza to, że przesunięcie w dziedzinie x jest mnożone przez 1 podczas poruszania się w tej przestrzeni. W bardziej ogólnym przypadku, zwłaszcza gdy operujemy na funkcjach wielowymiarowych, możemy poruszać się po różnych wektorach h. Należałoby je zatem uwzględnić we wzorze.

Definicja 2: Pochodna kierunkowa funkcji

Pochodną kierunkową wielowymiarowej funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ oraz $h \in \mathbb{R}^n : x + h \in \mathbb{D}$ w punkcie x wzdłuż wektora h nazwiemy funkcję:

$$\frac{df}{dh}(x, h) = \lim_{\delta \to 0} \frac{f(x+\delta h) - f(x)}{\delta} \tag{4.2}$$

Definicja 3: Pochodna kierunkowa funkcji (z wykorzystaniem gradientu)

Pochodną kierunkową wielowymiarowej funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ oraz $h \in \mathbb{R}^n : x + h \in \mathbb{D}$ w punkcie x wzdłuż wektora h nazwiemy funkcję:

$$\frac{df}{dh}(x, h) = \nabla_f(x)h = |\nabla_f(x)||h| \cos(\alpha) \tag{4.3}$$

gdzie $\nabla_f(x)$ jest gradientem funkcji f, a α jest kątem pomiędzy wektorami $\nabla_f(x)$ oraz h.

Pochodna cząstkowa jest szczególnym przypadkiem pochodnej kierunkowej. Pochodna cząstkowa opisuje szybkość, z jaką funkcja wielu zmiennych zmienia się względem jednej ze swoich zmiennych, przy jednoczesnym zachowaniu stałych pozostałych zmiennych. Zasadniczo jest to pochodna funkcji względem jednej z jej zmiennych niezależnych, traktująca pozostałe zmienne jako stałe. Pochodna cząstkowa jest także wrażliwością funkcji.

Definicja 4: Pochodna cząstkowa

Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ oraz $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Pochodną cząstkową funckji f w punkcie x wyliczaną po wartości x_i, $i = 1, 2, \ldots n$ nazwiemy funkcję:

$$\frac{\partial f}{\partial x_i}(x) = \frac{df}{de_i}(x)$$

gdzie e_i jest i-tym wersorem przestrzeni $\mathbb{R}^n$. Pochodna cząstkowa funckji f wyliczana po wartości x_i jest zatem a pochodną kierunkową f w kierunku wyznaczanym przez i-ty wersor, co oznacza że $h = e_i$.

4.2 Jądro (w algebrze liniowej)

Jądro w algebrze liniowej reprezentuje zbiór rozwiązań lub wektorów, które „znikają" lub są mapowane na wektor zerowy w ramach danej transformacji liniowej lub operacji macierzowej. Pojęcie jądra ma fundamentalne znaczenie w algebrze liniowej i posiada różne zastosowania, w tym wsparcie przy rozwiązywaniu układów równań liniowych i zrozumieniu właściwości przekształceń liniowych.

Rozważmy mapę liniową reprezentowaną jako macierz m × n o nazwie A ze współczynnikami w polu K, która operuje na wektorach kolumnowych x z n komponentami w odniesienu do K. The kernel of this linear map is the set of solutions to the equation $Ax = 0$, where 0 is understood as the zero vector.

$$\mathrm{N}(A) = \mathrm{Null}(A) = \ker(A) = \{\mathbf{x} \in K^n \mid A\mathbf{x} = \mathbf{0}\}. \tag{4.4}$$

Jądro przestrzeni n-wymiarowej jest macierzą tożsamościową o rozmiarze n. Mówiąc prościej, jest to macierz kwadratowa $n \times n$ z jedynkami na głównej przekątnej (od lewego górnego do prawego dolnego rogu) i zerami w pozostałych miejscach.

$$\ker(A_{n\times n}) = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}$$

Każdy wiersz tej macierzy (zwany także wersorem przestrzeni) jest oznaczany jako e_i, gdzie $i = 1, 2, \ldots n$. Wiersze te są przydatne przy obliczaniu pochodnej cząstkowej funkcji (patrz Definicja 4).

4.3 Optymalność przy pracy z przestrzenią wielowymiarową

Podczas rozwiązywania problemów związanych z funkcjami wielowymiarowymi, często mamy za zadanie znaleźć ekstremum tych funkcji. Jeśli używamy do tego metod lokalnych, powinniśmy wyznaczyć wektor, którego kierunek wskaże nam dane ekstremum. Metody lokalne, jak sama nazwa wskazuje, uwzględniają tylko najbliższe sąsiedztwo punktu w rozpatrywanej przestrzeni. Mając to na uwadze, najprostszym sposobem znalezienia ekstremum jest podążanie w kierunku najszybszego zmniejszania (lub zwiększania) funkcji. Wektor wskazujący w tym kierunku jest szukanym, optymalnym wektorem.

Optymalność wymaga od nas wcześniejszego określenia dwóch warunków:

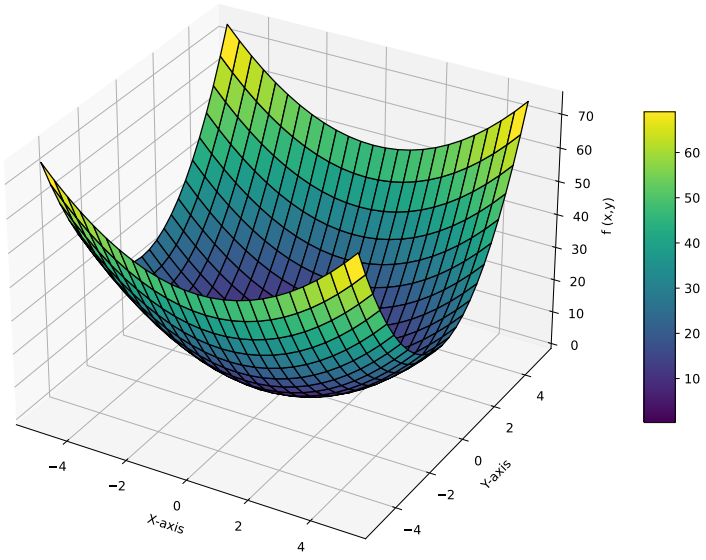
- Określenia rozpatrywanej funkcji oceny,

- Wyboru, czy chcemy zminimalizować, czy zmaksymalizować daną funkcję.

Funkcja oceny w większości przypadków jest po prostu badaną funkcją f. Możemy również zauważyć, że minimalizacja funkcji i maksymalizacja funkcji są bardzo podobnymi zadaniami. W rzeczywistości możemy zastąpić maksymalizację funkcji minimalizacją odwrotności tej funkcji.

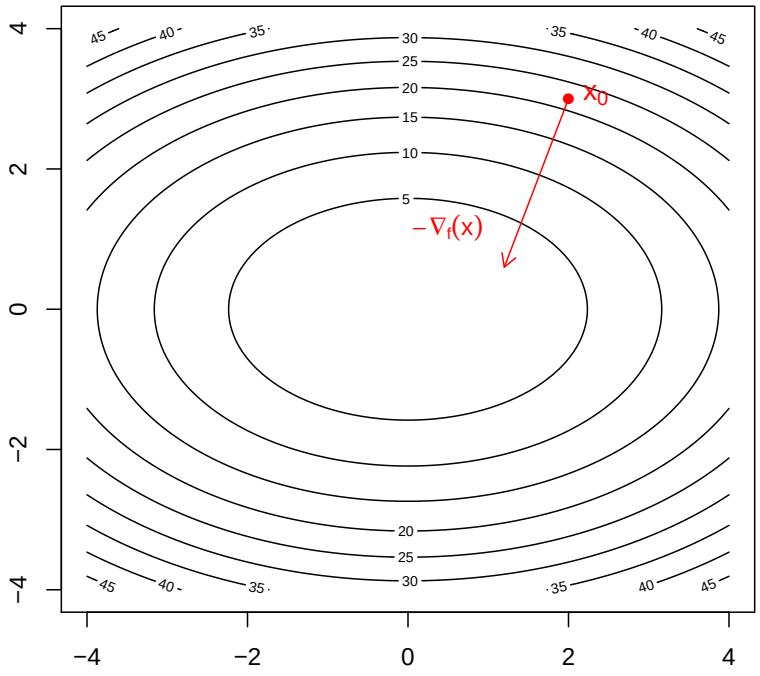
$$\operatorname{argmax} f = \operatorname{argmin}(-f) \tag{4.5}$$

Przykład 1. Niech $f(x, y) = x^2 + 2y^2$ oraz $x_0 = (2, 3)$. Wykres takiej funkcji wyglądałby następująco:



Rysunek 4.1: Funkcja $f(x) = x^2 + 2y^2$

Funkcja f posiada ekstremum w punkcie (0, 0). Gdybyśmy mieli znaleźć to ekstremum zaczynając ze startowego punktu x_0, musielibyśmy znaleźć optymalny wektor reprezentujący najszybszy spadek funkcji f. Wektor ten jest reprezentowany przez antygradient badanej funkcji $(-\nabla_f(x))$.

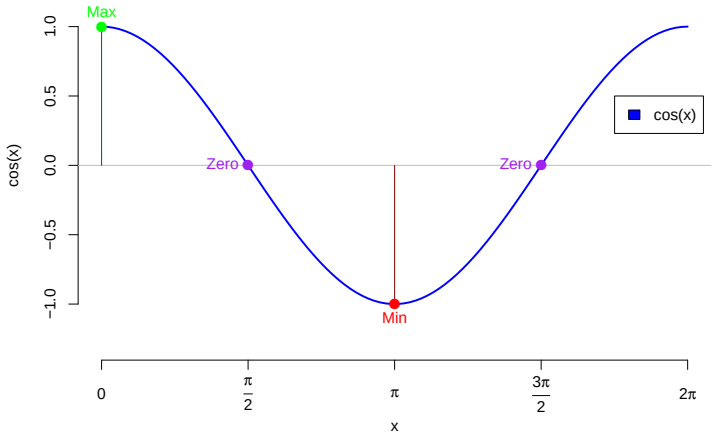


Rysunek 4.2: Wektor o punkcie zaczepienia x_0, który reprezentuje najszybszy spadek funkcji f

Pochodna kierunkowa może pomóc w znalezieniu optymalnego wektora. Jeśli chcielibyśmy zminimalizować funkcję (jak w przypadku Przykładu 1), należałoby wykorzystać wzór na pochodną kierunkową:

$$\underset{h}{\operatorname{argmin}} \frac{df}{dh}(x,h) = \underset{h}{\operatorname{argmin}} |\nabla_f(x)||h| \cos(\alpha) \tag{4.6}$$

Zakładając, że długość gradientu $\nabla_f(x)$ jest stała i długość wektora h jest również stała, głównym parametrem, którym możemy sterować, jest kąt pomiędzy tymi wektorami ($\cos(\alpha)$). Gradient wskazuje nam kierunek najszybszego wzrostu wartości funkcji. Naturalnie, jeśli chcemy zminimalizować funkcję, powinniśmy wybrać kierunek odwrotny - antygradient. Potwierdza to również wzór zamieszczony powyżej, ponieważ $\cos(180^o)$ jest równy -1, podczas gdy długości wektorów są zawsze wartościami dodatnimi.



Rysunek 4.3: Wykres funkcji cosinus

4.4 Metoda gradientu prostego (gradient descent)

Metoda gradientu prostego (*ang. gradient descent*) to algorytm optymalizacji powszechnie stosowany w celu minimalizacji nieliniowych funkcji analitycznych. W większości przypadków za funkcje te przyjmuje się funkcje kosztu lub straty w uczeniu maszynowym lub innych problemach optymalizacyjnych. Celem metody gradientu prostego jest iteracyjne dążenie do minimum funkcji poprzez dostosowywanie jej parametrów.

Definicja 5: Metoda gradientu prostego (gradient descent)

Biorąc pod uwagę, że metoda gradientu prostego jest algorytmem iteracyjnym, punktem obliczonym w $(k-1)$-tym kroku algorytmu na bazie funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ nazywamy:

$$x_{k+1} = x_k - \alpha \nabla_f(x_k) \tag{4.7}$$

gdzie $k \in \mathbb{N}$ jest numerem iteracji, α jest współczynnikiem długości kroków (*learning rate*), a $\nabla_f(x_k)$ jest gradientem funkcji f w punkcie x_k.

Ogólna koncepcja polega na powtarzaniu kroków w kierunku przeciwnym do gradientu (lub przybliżonego gradientu) funkcji w każdym kolejnym punkcie, ponieważ jest to kierunek najbardziej stromego opadania wartości funkcji. Z drugiej strony, krok w kierunku gradientu doprowadzi do lokalnego maksimum tej funkcji.

Przykład 2. Biorąc pod uwagę wzór na metodę gradientu prostego funkcji f, niech $x_1 = x_0 - \alpha \nabla_f(x_0)$, gdzie x_0 jest punktem startowym algorytmu, x_1 jest następnym punktem wyliczonym zgodnie z kierunkiem największego spadku funkcji oraz parametr $\alpha = 0.1$.

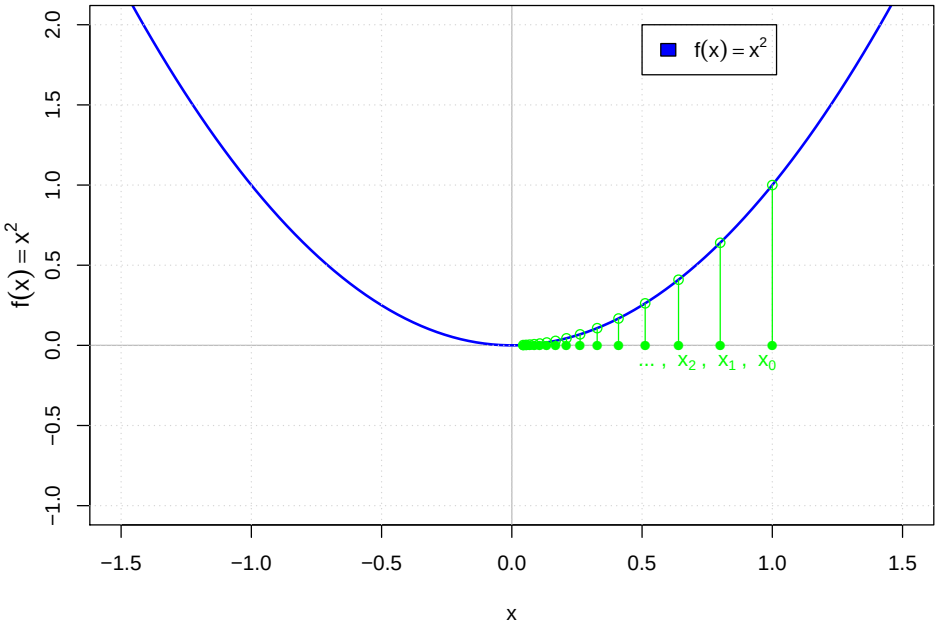
Postępując w ten sam sposób, możemy również obliczyć punkty możliwe do uzyskania w kolejnych iteracjach algorytmu:

- $x_2 = x_1 - \alpha \nabla_f(x_1)$
- $x_3 = x_2 - \alpha \nabla_f(x_2)$

- $x_4 = x_3 - \alpha\nabla_f(x_3)$ (...)

Niech $f(x) = x^2$ oraz $x_0 = 1$. Wartości kolejnych punktów x_0, x_1, x_2... w metodzie gradientu prostego wynoszą:

- $x_1 = x_0 - \alpha\nabla_f(x_0) = 1 - 0.1 \cdot 2 = 0.8$
- $x_2 = x_1 - \alpha\nabla_f(x_1) = 0.8 - 0.1 \cdot 1.6 = 0.64$
- $x_3 = x_2 - \alpha\nabla_f(x_2) = 0.64 - 0.1 \cdot 1.28 = 0.512$
- $x_4 = x_3 - \alpha\nabla_f(x_3) = 0.512 - 0.1 \cdot 1.024 = 0.4096$ (...)



Rysunek 4.4: Zbieganie kolejnych wartości punktów x_0, x_1, x_2... do ekstremum funkcji $f(x) = x^2$ po ustawieniu parametru $\alpha = 0.1$

Używając języka programowania **R** możemy napisać prostą implementację metody gradientu prostego. Przede wszystkim potrzebujemy funkcji do obliczenia gradientu. Możemy do tego użyć $grad(f, x)$ z biblioteki *numDeriv* lub możemy zaimplementować ją od zera w następujący sposób:

```
if(!require(docstring)) library(docstring) # Biblioteka do dokumentacji kodu!

num_grad <- function(f, x, h = 10^-6){
  #' Centralny Gradient Numeryczny
  #'
  #' @description Funkcja odpowiedzialna za wyznaczenie gradientu numerycznego
  #' poprzez obliczenie wektora srodkowych pochodnych czastkowych.
  #'
  #' @param f funkcja. Funkcja, ktorej gradient wyznaczamy.
  #' @param x wektor numeryczny. Punkt, w ktorym wyznaczany jest gradient
```

```
  #' numeryczny.
  #' @param h skalar. Skonczona wartosc roznicy.
  #'
  #' @usage num_grad(f, x)
  #'
  #' @return Wektor pochodnych czastkowych funkcji f.

  n <- length(x)
  g <- rep(NA, n) # wstepna alokacja pamieci do przechowywania gradientu
  e <- diag(n)

  # obliczanie pochodnych czastkowych
  for(i in 1 : n) g[i] = (f(x+h*e[i,])-f(x-h*e[i,]))/(2*h)
  return(g)
}
```

Listing 1: Implementacja gradientu numerycznego

Powyższa funkcja posiada komentarze umożliwiające podgląd dokumentacji funkcji. Dokumentacja ta zawiera krótki opis, parametry wejściowe i zwracane wartości. Dostęp do dokumentacji możemy uzyskać za pomocą następującego polecenia:

```
# Zapewnia dostep do dokumentacji
?num_grad
```

Listing 2: Funkcja umożliwiająca dostep do dokumentacji

Po uruchomieniu tego polecenia wyświetli się następująca dokumentacja:

Centralny Gradient Numeryczny

Description

Funkcja odpowiedzialna za wyznaczenie gradientu numerycznego poprzez obliczenie wektora srodkowych pochodnych czastkowych.

Usage

```
num_grad(f, x)
```

Arguments

f	funkcja. Funkcja, ktorej gradient wyznaczamy.
x	wektor numeryczny. Punkt, w ktorym wyznaczany jest gradient numeryczny.
h	skalar. Skonczona wartosc roznicy.

Value

Wektor pochodnych czastkowych funkcji f.

Rysunek 4.5: Dokumentacja docstring funkcji gradientu numerycznego

Teraz możemy przetestować działanie napisanej przez nas funkcji gradientu numerycznego. Aby to zrobić, najpierw musimy przygotować parametry wejściowe. Następnie możemy wywołać wspomnianą funkcję, sprawdzić jej wyniki i opcjonalnie przeprowadzić dodatkowe testy wydajnościowe.

```
# Parametry wejsciowe
my_fun <- function(x) 2*x[1]^2 + x[2]^2
```

```
c(3,4) -> x0  # Uwaga! Strzalki nie tylko w lewo!

# Wyniki
my_fun(x) # 2*3^2+4^2 = 17 # Sprobuj takze: sin(x^2);
x0 <- c(-3, -2)
my_fun(x0) # 2*(-3)^2+(-2)^2 = 22
num_grad(my_fun, x0, 10^-6) # zwraca wektor [-12, -4]

# Test 1
if(!require(numDeriv)) library(numDeriv)   # Biblioteka do obliczen numerycznych
grad(my_fun, x0)    # gradient
hessian(my_fun, x0) # hessian

# Test 2
if(!require(Deriv)) install.packages(Deriv); # Biblioteka do obliczen symbolicznych
my_fun <- function(x, y) 2*x^2 + y^2
df <- Deriv(my_fun)
cat('f =  ',  deparse(my_fun)[2], '\n')
cat('df = ', deparse(df)[2])
```

Listing 3: Testowanie gradientu numerycznego

Mając pod ręką działającą implementację funkcji gradientu, możemy przejść do tworzenia właściwej funkcji implementującej metodę gradientu prostego:

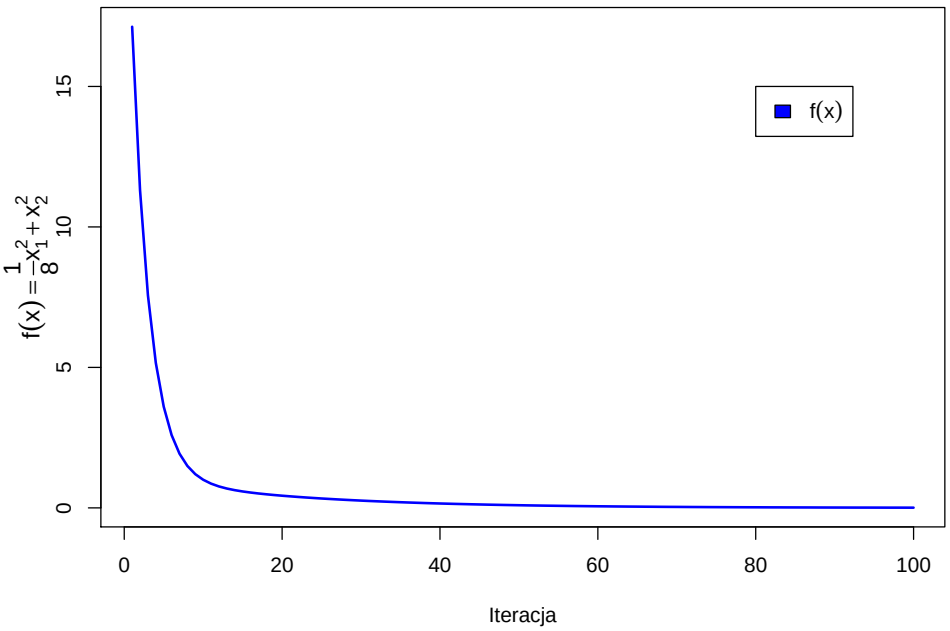
```
gradient_descent <- function(f, x, a = 0.1, K = 100){
  #' Metoda Gradientu Prostego
  #'
  #' @description Funkcja odpowiedzialna za obliczenie metody gradientu prostego
  #' dla funkcji f po K stopniach.
  #'
  #' @param f funkcja. Funkcja docelowa algorytmu.
  #' @param x wektor numeryczny. Punkt startowy algorytmu.
  #' @param a skalar. Opcjonalny parametr okreslajacy learning rate (domyslnie 0.1).
  #' @param K skalar. Opcjonalny parametr okreslajacy maksymalny limit iteracji (domyslnie
    100).
  #'
  #' @usage gradient_descent(f, x, a, K)
  #'
  #' @returns
  #' Lista wynikow zawierajaca nastepujace elementy:
  #' * x_opt: znalezione rozwiazanie,
  #' * f_opt: wartosc funkcji docelowej w znalezionym rozwiazaniu,
  #' * x_hist: historia badanych rozwiazan,
  #' * f_hist: historia wartosci funkcji docelowej,
  #' * t_eval: czas, jaki uplynal podczas obliczen algorytmu.

  start_time <- Sys.time()
  results <- list(x_opt = x,
                  f_opt = f(x),
                  x_hist = matrix(NA, nrow = K, ncol = length(x)),
                  f_hist = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)

  for(k in 2:  K){
    # opis przejscia od punktu x_k do x_k+1
    x_new <- x - a * grad(f, x)

    # sprawdzenie, czy nowe rozwiazanie
    # jest najlepsze do tej pory
    if(f(x_new) < results$f_opt){
```

```
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }

    results$x_hist[k,] <- x_new
    results$f_hist[k] <- f(x_new)

    x <- x_new
  }
  # roznica czasu pomiedzy koncem i poczatkiem algorytmu
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 4: Implementacja metody gradientu prostego

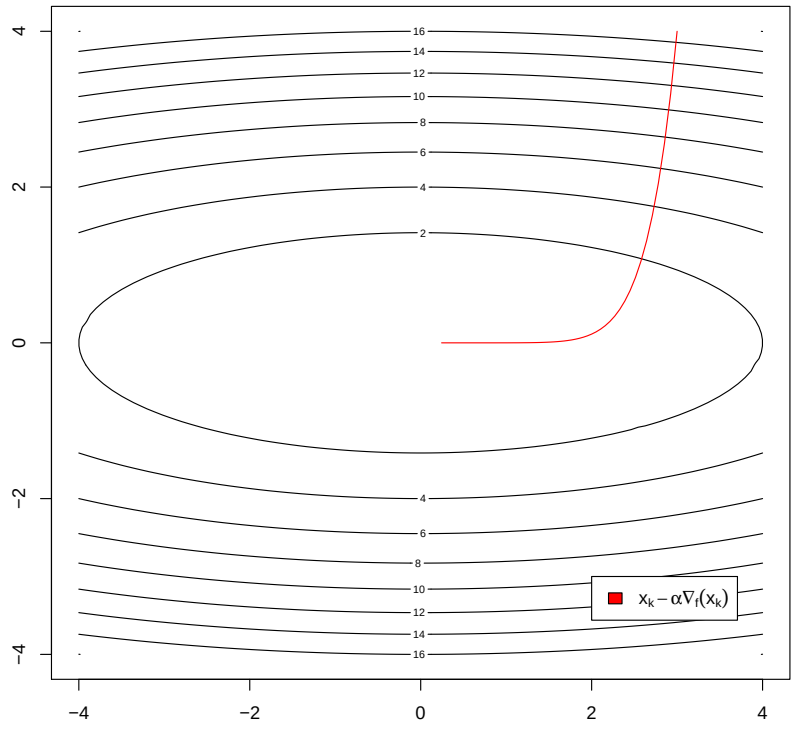
Przykład 3. Niech $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ oraz $x_{start} = (3, 4)$.

Funkcja f w punkcie x_{start} przyjmuje wartość $17\frac{1}{8}$. Po wykonaniu $K = 100$ iteracji napisanej przez nas implementacji metody gradientu prostego wartość ta spadła do około 0.00711, dzięki czemu jest o wiele bliższa globalnemu minimum funkcji f - czyli zeru. Uzyskane wartości funkcji f na każdym etapie tego algorytmu można zobaczyć na poniższym wykresie:



Rysunek 4.6: Wartości funkcji f stopniowo uzyskiwane podczas 100 iteracji metody gradientu prostego

Możemy wykreślić nie tylko wartości obliczone przez ten algorytm, ale także pozycje uzyskane w kolejnych iteracjach w przestrzeni 2D. Ścieżkę pokonaną przez algorytm metody gradientu prostego można zobaczyć na poniższym wykresie:



Rysunek 4.7: Ścieżka przebyta przez algorytm metody gradientu prostego funkcji f w ciągu 100 iteracji w przestrzeni 2D

4.5 Learning rate

Learning rate to hiperparametr, który odgrywa kluczową rolę w kontrolowaniu wielkości kroku w każdej iteracji algorytmu metody gradientu prostego. W kontekście uczenia maszynowego learning rate jest często reprezentowana przez symbol α. Współczynnik learning rate określa, jak bardzo powinniśmy dostosować parametry względem gradientu funkcji kosztu.

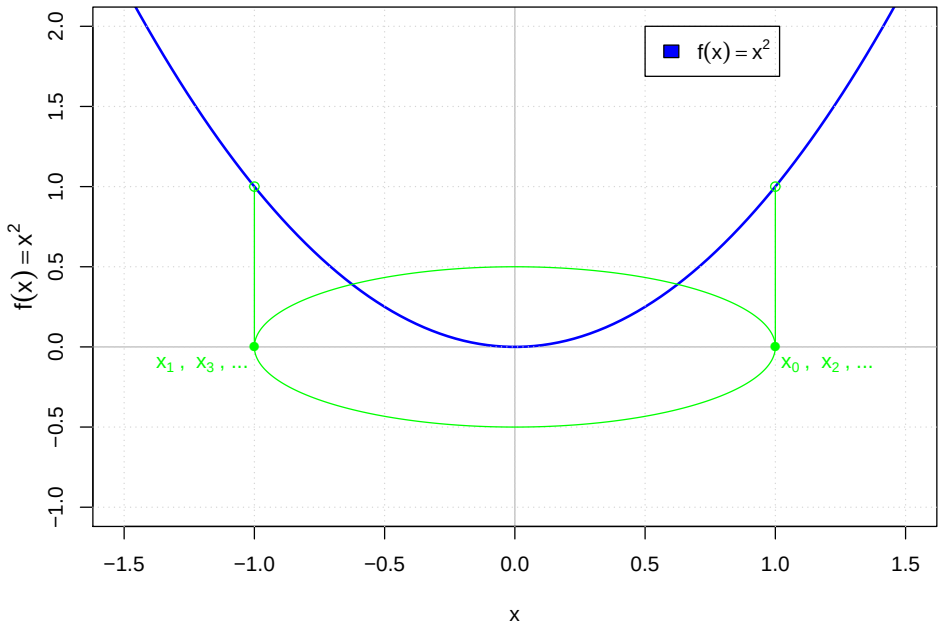
Dlaczego i jak się go używa? Rozważmy następujący przykład:

> **Przykład 4.** Wiedząc, że gradient wskazuje na największy wzrost funkcji f, niech $x_1 = x_0 - \nabla_f(x_0)$, gdzie x_0 jest punktem startowym algorytmu oraz x_1 jest kolejnym punktem obliczonym w kierunku największego spadku funkcji. Na razie załóżmy, że learning rate α jest równy 1.
>
> Postępując w ten sam sposób, możemy obliczyć punkty w kolejnych iteracjach w podobny sposób:
>
> - $x_2 = x_1 - \nabla_f(x_1)$
> - $x_3 = x_2 - \nabla_f(x_2)$
> - $x_4 = x_3 - \nabla_f(x_3)$ (...)

Niech $f(x) = x^2$ oraz $x_0 = 1$. Wiedząc, że gradient funkcji jednowymiarowej jest prostą pochodną tej funkcji, możemy obliczyć punkty $x_0, x_1, x_2...$ dla $f(x)$:

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 2 = -1$
- $x_2 = x_1 - \nabla_f(x_1) = -1 - (-2) = 1$
- $x_3 = x_2 - \nabla_f(x_2) = 1 - 2 = -1$
- $x_4 = x_3 - \nabla_f(x_3) = -1 - (-2) = 1$ (...)

Jak widać, po tak zdefiniowanych krokach osiągnięto swego rodzaju impas. Kolejne wartości oscylują wokół ekstremum, nigdy się do niego nie zbiegając.



Rysunek 4.8: Impas powstały w wyniku wykorzystania całej wartości gradientu przy obliczaniu punktów $x_0, x_1, x_2...$

Sytuacja jeszcze się pogorszy, jeśli założymy, że funkcja $f(x) = x^4$. Obliczając punkty $x_0, x_1, x_2...$ otrzymujemy coraz większe wartości:

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 4 = -3$
- $x_2 = x_1 - \nabla_f(x_1) = -3 - (-108) = 105$
- $x_3 = x_2 - \nabla_f(x_2) = 105 - 4630500 = -4630395$
- $x_4 = x_3 - \nabla_f(x_3) \approx -4630395 - (-3.9711301e + 20) \approx -3.9711301e + 20$ (...)

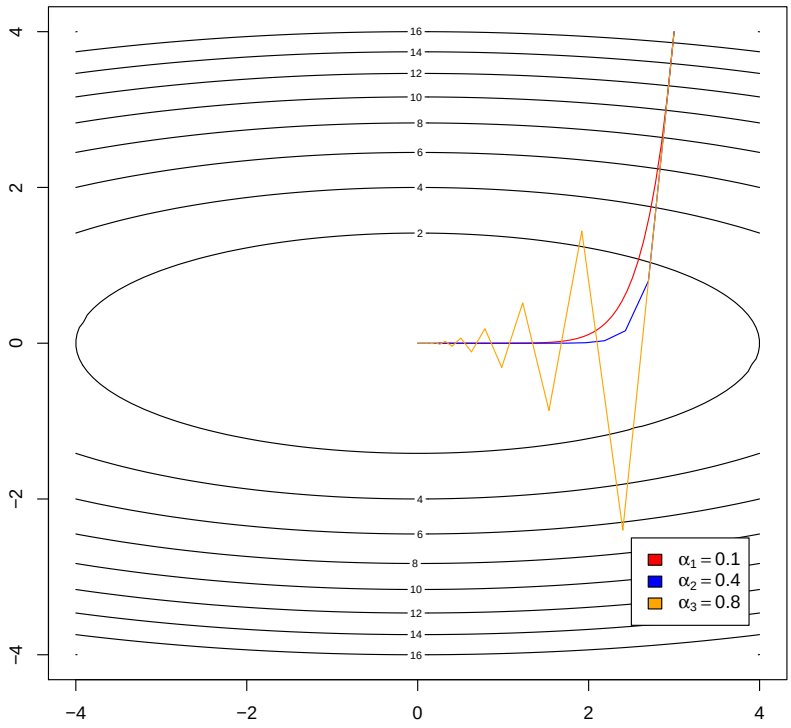
Jednym z możliwych rozwiązań problemu przedstawionego w Przykładzie 4 jest wprowadzenie pewnego rodzaju parametru uczenia się, parametru przestrzeni uczenia się lub parametru wielkości kroku (jakkolwiek

chcemy go nazwać). Ten parametr będzie odpowiedzialny za kontrolowanie rozmiaru kroków, które wykonujemy w każdej iteracji. Innymi słowy, byłby odpowiedzialny za zmniejszanie wpływu gradientu na każde z obliczeń. Wspomniany parametr jest dokładnie tym, czego oczekujemy od learning rate. Jest to integralna część metody gradientu prostego i ma duży wpływ na działanie tego algorytmu.

Learning rate to ważny hiperparametr, który należy starannie wybrać. Jeśli jest on zbyt mały, algorytm może osiągać zbieżność bardzo powoli, wymagając dużej liczby iteracji w celu osiągnięcia minimum. Z drugiej strony, jeśli szybkość uczenia się jest zbyt duża, algorytm może przeskoczyć minimum i nie osiągnąć zbieżności lub oscylować wokół minimum.

Przykład 5. Niech $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ oraz $x_{start} = (3, 4)$.

Ścieżka przebyta przez metodę gradientu prostego może się różnić w zależności od wybranej wartości parametru learning rate. Niech $\alpha_1 = 0.1, \alpha_2 = 0.4$ oraz $\alpha_3 = 0.8$. Jak metoda gradientu prostego zachowuje się przy użyciu tych parametrów, można zobaczyć na poniższym wykresie:



Rysunek 4.9: Wpływ learning rate na ścieżkę przebytą przez metodę gradientu prostego na funkcji f.

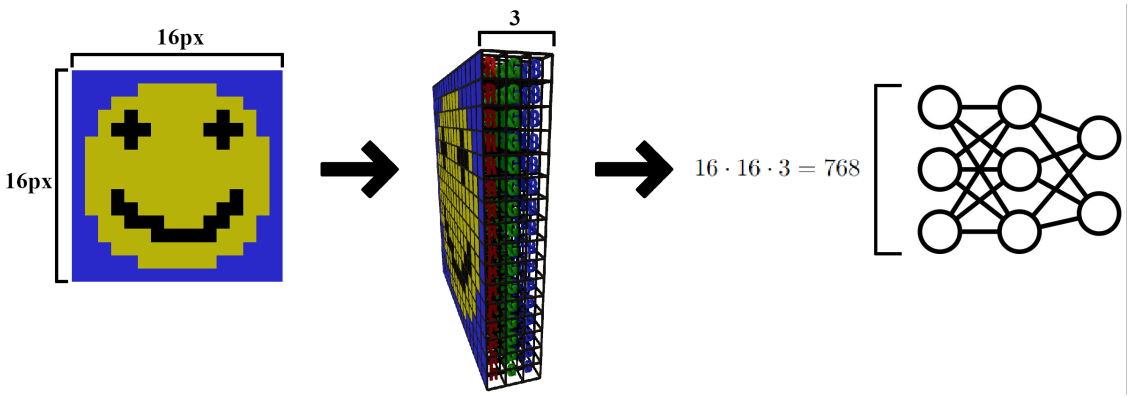
4.6 Metoda gradientu prostego w sieciach neuronowych

Metoda gradientu prostego jest często używana przy pracy z sieciami neuronowymi. Jest to podstawowy algorytm optymalizacyjny leżący u podstaw uczenia sieci neuronowej. Celem uczenia jest minimalizacja

funkcji kosztu, która mierzy różnicę między przewidywanymi wynikami sieci neuronowej a rzeczywistymi wartościami docelowymi.

Algorytm ten dobrze nadaje się do użycia podczas optymalizacji funkcji wieloargumentowych. Problemy, do rozwiązywania których wykorzystuje się sieci neuronowe, często dotyczą właśnie takich funkcji. Jednym z typowych zadań, do jakich wykorzystuje się sieci neuronowe, jest między innymi klasyfikacja obrazów.

Tworząc sieć neuronową, należy określić jej architekturę, w tym liczbę warstw, liczbę neuronów w każdej warstwie oraz funkcje aktywacji. Pierwsza warstwa neuronów, będąca warstwą wejściową, jest bezpośrednio powiązana z rodzajem danych wejściowych. W przypadku klasyfikacji obrazów liczba neuronów w tej warstwie jest zwykle równa liczbie pikseli analizowanego obrazu pomnożonej przez 3 przy uwzględnieniu kolorów (dla każdego z kanałów RGB).



Rysunek 4.10: Liczba pikseli pomnożona przez 3 kanały RGB jako dane wejściowe dla sieci neuronowej

Oprócz warstwy wejściowej musimy również określić liczbę neuronów w warstwach ukrytych i warstwie wyjściowej. Liczba i struktura warstw ukrytych może się różnić w zależności od wybranego podejścia do budowy sieci neuronowej. Liczba neuronów w warstwie wyjściowej odpowiada jednak zwykle liczbie możliwych wyników – kategorii.

Każdy neuron w sieci ma połączenia z sąsiednimi warstwami. Połączenia te, podobnie jak neurony, mają specjalne parametry zwane wagami. Wagi reprezentują siłę połączeń między neuronami w różnych warstwach sieci neuronowej. Każde połączenie między neuronami jest powiązane z wagą, która jest dostosowywana podczas procesu uczenia, aby umożliwić sieci dokonywanie dokładnych predykcji. Istnieją również dodatkowe parametry, zwane bias, które umożliwiają sieci przesunięcie funkcji aktywacji. Zapewniają modelowi elastyczność pozwalającą uwzględnić sytuacje, w których sygnał wejściowy neuronu jest niewystarczający, aby go aktywować. Bias-y są dostosowywane podczas uczenia wraz z wagami, aby poprawić ogólną wydajność sieci. W warstwie sieci neuronowej wyjście każdego neuronu oblicza się poprzez zastosowanie ważonej sumy wejść, po której następuje funkcja aktywacji. Matematycznie wynik O_j neuronu j w warstwie jest określony wzorem:

$$O_j = \sigma\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right) \tag{4.8}$$

gdzie w_{ij} jest wagą połączenia między neuronem i w warstwie poprzedniej oraz neuronem j w bieżącej warstwie, x_i jest wejściem z neuronu i, b_j jest biasem neuronu j, σ jest funckją aktywacji, a n jest liczbą neuronów w poprzedniej warstwie.

Wagi i bias-y są początkowo wybierane losowo, ale należy je dostosować podczas uczenia sieci neuronowej. Jak można się spodziewać, wyniki sieci inicjowanych losowo w większości przypadków nie są dobre. Aby oszacować skuteczność sieci neuronowej, tworzona jest tak zwana *funkcja kosztu*. Jest to miara matematyczna, która określa różnicę między przewidywaną wartością wyjściową sieci a rzeczywistymi wartościami docelowymi. Popularna funkcja kosztu używana w problemach regresyjnych, gdzie celem jest przewidzenie wartości ciągłej, nazywana jest *błędem średniokwadratowym* (MSE). Błąd średniokwadratowy to średnia kwadratów różnic między wartościami przewidywanymi i rzeczywistymi:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.9}$$

gdzie n jest liczbą danych, y_i jest wartością docelową, a $\hat{y}_i$ jest uzyskaną wartością wyjściową.

Możemy myśleć o funkcji kosztu jako o funkcji, która jako dane wejściowe przyjmuje wszystkie wagi i bias-y sieci, a na wyjściu otrzymuje jedną liczbę stanowiącą ocenę wydajności sieci. Ta funkcja wieloparametrowa jest następnie minimalizowana za pomocą metody gradientu prostego. Wektor utworzony za pomocą metody gradientu prostego zawiera szereg zmian, jakie powinny nastąpić w każdej z wag i bias-ów, aby zbliżyć się do minimum funkcji kosztu – tak aby wyniki uzyskane przez sieć były bliższe oczekiwanym wynikom.

$$\nabla C(w_0, w_1, ..., w_n) \tag{4.10}$$

gdzie ∇C jest gradientem użytym w algorytmie, n jest sumą wszystkich wag i bias-ów oraz w_n jest wartością n-tej wagi.

4.7 Metoda najszybszego spadku (steepest descent)

Metoda najszybszego spadku (*ang. steepest descent*) to kolejny algorytm optymalizacji powszechnie stosowany w celu minimalizacji nieliniowych funkcji analitycznych. W swojej strukturze jest bardzo podobny do metody gradientu prostego. Celem metody najszybszego spadku jest również iteracyjne dążenie do minimum funkcji poprzez dostosowanie jej parametrów. Jednak w przeciwieństwie do metody gradientu prostego nie ustawiamy z góry parametru *learning rate* algorytmu. Podajemy jedynie maksymalną wartość *learning rate* (maksymalną wielkość kroku w kierunku antygradientu), a następnie algorytm sam określa optymalną wartość tego parametru.

Definicja 6: Metoda najszybszego spadku (steepest descent)

Biorąc pod uwagę, że metoda najszybszego spadku jest algorytmem iteracyjnym, punktem obliczonym w $(k-1)$-tym kroku algorytmu na bazie funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ nazywamy:

$$x_{k+1} = x_k - \alpha_{k-best}\nabla_f(x_k) \tag{4.11}$$

gdzie $k \in \mathbb{N}$ jest numerem iteracji, $\nabla_f(x_k)$ jest gradientem funkcji f w punkcie x_k, a α_{k-best} jest współczynnikiem *learning rate* znalezionym podczas g kroków przeszukiwania liniowego.

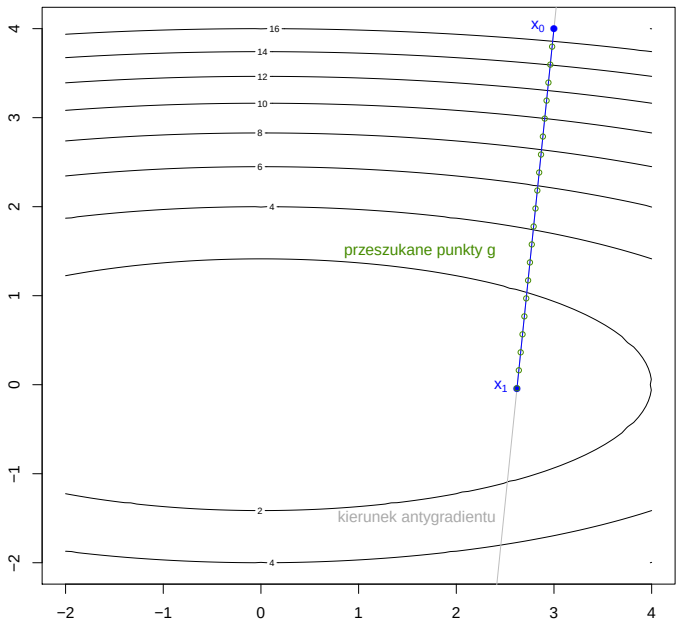
Korzystając z tego algorytmu, nie tylko podejmujemy kroki w optymalnym kierunku, ale także automatycznie wybieramy najlepszy rozmiar tych kroków. Najlepszy rozmiar kroku lub *learning rate* możemy obliczyć na kilka sposobów. Jednym ze sposobów jest skorzystanie ze wzoru na metodę gradientu prostego i użycie go do obliczenia jego pochodnej po *learning rate*, czyli α. Znalezienie minimum tej pochodnej dałoby nam optymalny parametr α.

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} \frac{d}{d\alpha} \left(x_k - \alpha \nabla_f(x_k)\right) \tag{4.12}$$

Niestety, z tym rozwiązaniem jest pewien problem. Gdybyśmy w naszym równaniu użyli pochodnej, otrzymalibyśmy algorytm symboliczny. Implementacja takiego algorytmu wymagałaby od nas umiejętności szybkiego obliczenia pochodnej dowolnej funkcji. Nie zawsze jest to łatwe zadanie. Komputerom łatwiej jest wykonywać obliczenia przy użyciu algorytmów numerycznych. To prowadzi nas do kolejnego rozwiązania – przybliżenia optymalnej wartości *learning rate*.

Aby przybliżyć wielkość kroku, możemy przeszukać liniowo różne punkty wzdłuż kierunku wskazywanego przez antygradient i sprawdzić, jakie wyniki dzięki nim uzyskamy. Oczywiście nie chcemy przeszukiwać nieskończonej liczby punktów, jakie możemy znaleźć na tej prostej. W tym celu należy ustawić maksymalny zasięg wyszukiwania (czyli górną granicę α - α_{max}) oraz punkt początkowy wyszukiwania x_k. Jednakże na odcinku znajduje się również nieskończona liczba punktów. Dlatego też ustawiamy również parametr g, który określa ile punktów powinniśmy sprawdzić na tym odcinku.

Metoda najszybszego spadku przypomina algorytm metody gradientu prostego z jedną kluczową różnicą. W każdym kroku algorytmu wyznaczamy punkty g w kierunku antygradientu i ustawiamy je w równej odległości od siebie na odcinku długości kontrolowanym przez α_{max}. Jest to równoznaczne z dyskretnym przejściem przez g kroków od $\alpha = 0$ do $\alpha = \alpha_{max}$ i wybraniem najlepszego wyniku.



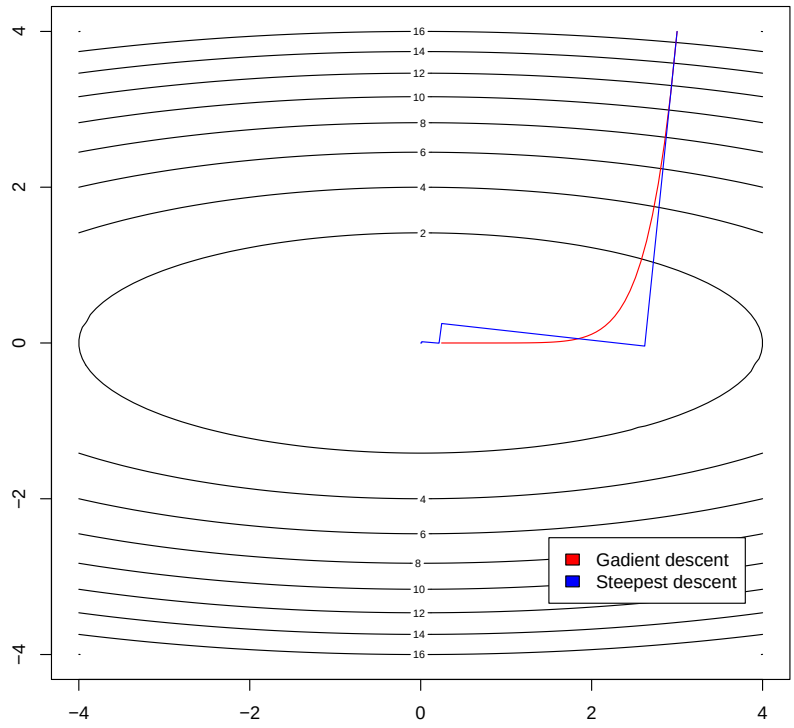
Rysunek 4.11: Punkty badane podczas jednej iteracji metody najszybszego spadku na funkcji f

Jest to podejście w pewnym sensie heurystyczne. Algorytm ten nie poda nam optymalnej wartości wielkości kroku, lecz jej przybliżenie. Jednakże, przy odpowiednio dobranym parametrze g uzyskamy wartość bliską optymalnej przy relatywnie niskim koszcie obliczeniowym. Główną zaletą tego algorytmu w porównaniu z metodą gradientu prostego jest lepiej dostosowana wielkość kroku w każdej iteracji. Niestety ma to też swoje

wady – m.in. wyższe koszty obliczeniowe pojedynczej iteracji, co jednak często w dłuższej perspektywie jest rekompensowane lepszą jakością uzyskanych kroków.

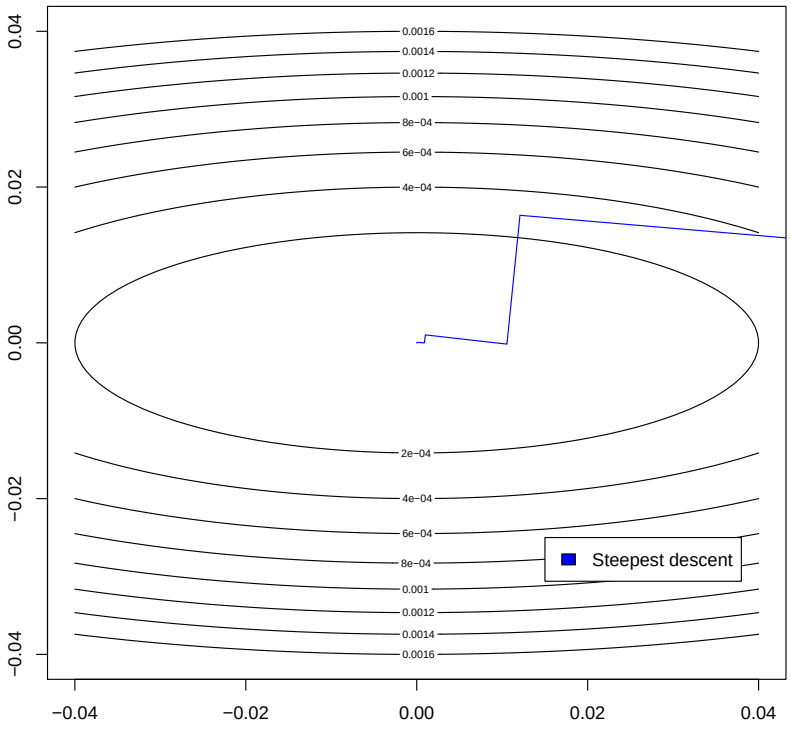
Przykład 6. Niech $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ oraz $x_{start} = (3, 4)$.

Dodatkowo niech parametrami metody najszybszego spadku będą $\alpha_{max} = 5$ i $g = 1000$. Jak algorytm zachowuje się przy użyciu tych parametrów, można zobaczyć na poniższym wykresie (wraz ze ścieżką utworzoną przez metodę gradientu prostego):



Rysunek 4.12: Porównanie ścieżki uzyskanej przez metodę najszybszego spadku i metodę gradientu prostego na funkcji f w przestrzeni 2D

Możemy zaobserwować interesujący wzór na ścieżce utworzonej przez algorytm metody najszybszego spadku. Kolejne kroki iteracji algorytmu tworzą odcinki prostopadłe do siebie. Ten sam wzór się powtarza i coraz bardziej zbliża się do ekstremum funkcji. To zachowanie nie jest losowe ani przypadkowe. Wynika ono z faktu, że gradient funkcji w dowolnym punkcie nie musi wskazywać ekstremum globalnego, ale kierunek najszybszego spadku wartości funkcji. Wielkość kroku jest dostosowywana liniowo, aż funkcja przestanie maleć i osiągnie punkt stacjonarności. Dociera wówczas do przestrzeni stycznej, gdzie funkcja przestaje maleć (a w miarę dalszego ruchu zaczyna rosnąć). Od nowo wyznaczonego punktu, gdyż leży on na przestrzeni stycznej, kierunek najszybszego spadku funkcji leży pod kątem 90^o od kierunku ostatniego kroku.



Rysunek 4.13: Zbliżenie ścieżki uzyskanej przez metodę najszybszego spadku na funkcji f w przestrzeni 2D

Z wykorzystaniem języka programowania **R** możemy napisać własną implementację metody najszybszego spadku. Po pierwsze potrzebujemy funkcji, która przeszuka liniowo najlepsze rozwiązanie na przestrzeni g punktów:

```
line_search <- function(f, x0, x1, g = 100) {
  #' Przeszukiwanie Liniowe
  #'
  #' @description Funkcja pomocnicza odpowiedzialna za znalezienie najlepszego punktu
  #' na podstawie funkcji f sposrod g punktow o rozkladzie liniowym.
  #'
  #' @param f funkcja. Funkcja docelowa algorytmu.
  #' @param x0 wektor numeryczny. Punkt startowy algorytmu.
  #' @param x1 wektor numeryczny. Punkt koncowy algorytmu (lub maksymalny zakres kroku).
  #' @param g skalar. Parametr opcjonalny, ktory odpowiada za liczbe iteracji wyszukiwania
    najlepszego rozmiaru kroku
  #' w kazdej iteracji algorytmu (domyslnie jest to 100).
  #'
  #' @usage line_search(f, x0, x1, g)
  #'
  #' @returns
  #' x_best: najlepszy znaleziony punkt na podstawie funkcji f

    # ustawienie x0 jako punktu poczatkowego
    x_best <- x0
    # petla po punktach g w kierunku punktu x1
    for(i in 1 : g) {
```

```
        t <- i / g
        x_t <- t*x1+(1-t)*x0
        if(f(x_t) < f(x_best)) {
            x_best <- x_t
        } else {
            break
        }
    }
    return(x_best)
}
```

Listing 5: Implementacja przeszukiwania liniowego

Mając już gotową funkcję wyszukiwania liniowego, możemy przystąpić do implementacji głównej części algorytmu. Przykładowo, można to zrobić jako funkcję *steepest descent* opartą na poprzedniej implementacji metody gradientu prostego:

```
steepest_descent <- function(f, x, a = 5, g = 100, K = 100){
  #' Metoda Najszybszego Spadku
  #'
  #' @description Funkcja odpowiedzialna za obliczenie metody najszybszego spadku
  #' funkcji f dla g punktow w kazdej iteracji po K krokach.
  #'
  #' @param f funkcja. Funkcja docelowa algorytmu.
  #' @param x wektor numeryczny. Punkt startowy algorytmu.
  #' @param a skalar. Opcjonalny parametr okreslajacy maksymalny learning rate (domyslnie 5)
    .
  #' @param g skalar. Parametr opcjonalny, ktory odpowiada za liczbe iteracji wyszukiwania
    najlepszego rozmiaru kroku
  #' w kazdej iteracji samego algorytmu (domyslnie jest to 100).
  #' @param K skalar. Opcjonalny parametr okreslajacy maksymalny limit iteracji algorytmu (
    domyslnie 100).
  #'
  #' @usage steepest_descent(f, x, a, g, K)
  #'
  #' @returns
  #' Lista wynikow zawierajaca nastepujace elementy:
  #' * x_opt: znalezione rozwiazanie,
  #' * f_opt: wartosc funkcji docelowej w znalezionym rozwiazaniu,
  #' * x_hist: historia badanych rozwiazan,
  #' * f_hist: historia wartosci funkcji docelowej,
  #' * t_eval: czas, jaki uplynal podczas obliczen algorytmu.

  start_time <- Sys.time()
  results <- list(x_opt = x,
                  f_opt = f(x),
                  x_hist = matrix(NA, nrow = K, ncol = length(x)),
                  f_hist = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x
  results$f_hist[1] <- f(x)

  for(k in 2:  K){
    # opis przejscia od punktu x_k do x_k+1
    x_new <- line_search(f, x, x - a * grad(f, x), g)

    # sprawdzenie, czy nowe rozwiazanie
    # jest najlepsze do tej pory
    if(f(x_new) < results$f_opt){
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }

```

```
45      results$x_hist[k,] <- x_new
46      results$f_hist[k] <- f(x_new)
47
48      x <- x_new
49    }
50    # roznica czasu pomiedzy koncem i poczatkiem algorytmu
51    results$t_eval <- Sys.time() - start_time
52    return(results)
53  }
```

Listing 6: Implementacja metody najszybszego spadku