131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 1: Optymalizacja analityczna bez ograniczeń

Daniel Kaszyński

Kwestie organizacyjne

- **Prowadzący zajęcia**: mgr Daniel Kaszyński, dkaszy[@]sgh.waw.pl
- **Konsultacje**: wtorek 12:00-13:00, G-224.
- **Tryb zaliczenia przedmiotu**: egzamin pisemny w sesji z treści przedstawianych w trakcie zajęć lub zawartych w materiałach.

  Przykłady zadań: opisz metodę XYZ, wskaż na różnice między metodami ABC i XYZ, rozwiąż przedstawione zadanie optymalizacji analitycznej, przeprowadź dwie iteracje metody XYZ.
- **Wymagana literatura**:
  - [KW19] Kochenderfer, M.J. and Wheeler, T.A., 2019. *Algorithms for optimization.* Mit Press.
  - [CZ04] Chong, E.K. and Zak, S.H., 2004. *An introduction to optimization.* John Wiley & Sons.
  - [BI86] Birkholc, A., 1986. *Analiza matematyczna: funkcje wielu zmiennych.* Państwowe Wydawnictwo Naukowe.
  - [SS08] Sydsæter, K., Hammond, P., Seierstad, A. and Strom, A., 2008. *Further mathematics for economic analysis.* Pearson education.
  - [CO14] Cortez, P., 2014. *Modern optimization with R.* New York: Springer.

1.1 Definicja ekstremum

Terminem centralnym w zakresie optymalizacji analitycznej jest pojęcie ekstremum: rozwiązania, które dla zdefiniowanej funkcji oceniającej, tzw. **funkcji celu**, najczęściej oznaczanej jako f, generuje 'najlepszą' wartość. Rozważmy przykład $f : \mathbb{R} \to \mathbb{R}$.

Definicja 1: Definicja lokalnego ekstremum właściwego (minimum)

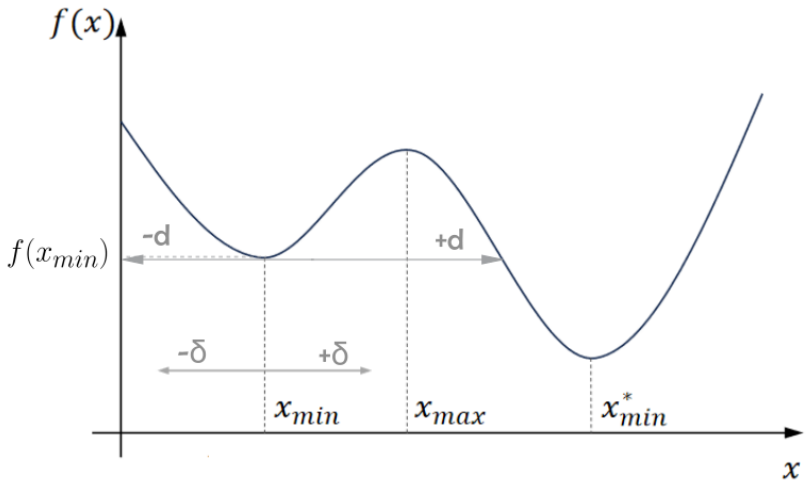
Lokalnym ekstremum właściwym (minimum) nazwiemy punkt x^*, który:

$$\exists\, d > 0 \quad \forall\, 0 < |\delta| < |d| \quad \Rightarrow \quad f(x^* + \delta) \geqslant f(x^*) \tag{1.1}$$

Definicja 2: Definicja globalnego ekstremum właściwego (minimum)

Globalnym ekstremum właściwym (minimum) nazwiemy punkt x^*, który:

$$\forall\, d > 0 \quad \forall\, |\delta| > 0 \quad \Rightarrow \quad f(x^* + \delta) > f(x^*) \tag{1.2}$$



?figurename? 1.1: Ekstremum lokalne funkcji $f : \mathbb{R} \to \mathbb{R}$

Zauważmy, że różnica między ekstremum *lokalnym* i *globalny* to obszar, w którym uzyskujemy *lepsze* (tj. z perspektywy funkcji celu, funkcji oceniającej) rozwiązanie. W przypadku ekstremum lokalnego, można wskazać otoczenie (może być to bardzo mały obszar!) sąsiedztwa w okół ekstremum x_{min}, w którym utrzymujemy stricte *gorsze* rozwiązania. W przypadku ekstremum *globalnego* x^*_{min}, takie sąsiedztwo to dowolne otoczenie rozwiązania ekstremum.

1.2 Optymalizacja nieliniowa bez ograniczeń, przypadek *jednowymiarowy*, tj. $f : \mathbb{R} \to \mathbb{R}$

W zakresie podstawowych twierdzeń, musimy wpierw wskazać podstawową dla nas definicję – definicję pochodnej funkcji jednej zmiennej.

Definicja 3: Pochodna funkcji

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

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{1.3}$$

Przykład 1. Przyjmijmy $f(x) = x^2 + 2x$. Wyznacz pochodną funkcji w punkcie $x_0 = 2$ z definicji:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \lim_{h \to 0} \frac{(x+h)^2 + 2(x+h) - x^2 - 2x}{h} = \tag{1.4}$$

$$= \lim_{h \to 0} \frac{x^2 + 2xh + h^2 + 2x + 2h - x^2 - 2x}{h} = \lim_{h \to 0} \frac{2xh + h^2 + 2h}{h} = \tag{1.5}$$

$$= \lim_{h \to 0} 2x + h + 2 = 2x + 2 \tag{1.6}$$

$$f'(2) = 2 \times 2 + 2 = 6 \tag{1.7}$$

W analogiczny sposób można wyprowadzić wzór na pochodną dowolnego rzędy:

$$f''(x) = \frac{d^2 f}{dx^2}(x) = (f'(x))' = \lim_{h \to 0} \frac{f'(x+h) - f'(x)}{h} = \lim_{h \to 0} \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} \tag{1.8}$$

1.2.1 Warunki Pierwszego Rzędu $f : \mathbb{R} \to \mathbb{R}$

Twierdzenie 1: Warunki Pierwszego Rzędu $f : \mathbb{R} \to \mathbb{R}$.
Niech $f : \mathbb{D} \subset \mathbb{R} \to \mathbb{R}$, $f \in C^1$. Jeżeli funkcja f posiada ekstremum w punkcie $x \in \mathbb{D}$, to $f'(x) = 0$.

?proofname? Ponieważ funkcja f ma w punkcie x minumim, wiemy, że istnieje $d > 0$, takie, że dla każdego $0 < |\delta| < d$, mamy $f(x+\delta) > f(x)$, czyli $f(x+\delta) - f(x) > 0$. Dzieląc obie strony przez δ, otrzymujemy:

$$\text{dla } \delta > 0 \qquad \frac{f(x+\delta) - f(x)}{\delta} > 0 \qquad \wedge \qquad \text{dla } \delta < 0 \qquad \frac{f(x+\delta) - f(x)}{\delta} < 0$$

Odpowiednio przy przejściu granicznym $\delta \to 0^+$ i $\delta \to 0^-$ mamy:

$$\lim_{\delta \to 0^+} \frac{f(x+\delta) - f(x)}{\delta} = f'_+(x) \geqslant 0 \qquad \wedge \qquad \lim_{\delta \to 0^-} \frac{f(x+\delta) - f(x)}{\delta} = f'_-(x) \leqslant 0$$

Jeśli funkcja f jest różniczkowalna, mamy $f'(x) = f'_+(x) = f'_-(x) = 0$. □

Warunki Pierwszego Rzędu dają nam możliwość przefiltrowania przestrzeni rozwiązań, umożliwiając selekcję rozwiązań, w których pochodna funkcji jest równa zero (tzw. punkty *stacjonarne*). **Uwaga!** To, że w danym punkcie mamy $f'(x) = 0$ nie oznacza, że w tym punkcie występuje ekstremum funkcji. Jako przykład rozważ funkcje $f(x) = x^2$ oraz $f(x) = x^3$.

1.2.2 Twierdzenie Taylora $f : \mathbb{R} \to \mathbb{R}$

Żebyśmy mogli ruszyć dalej, potrzebne nam jest jedno z istotniejszych twierdzeń analizy matematycznej!

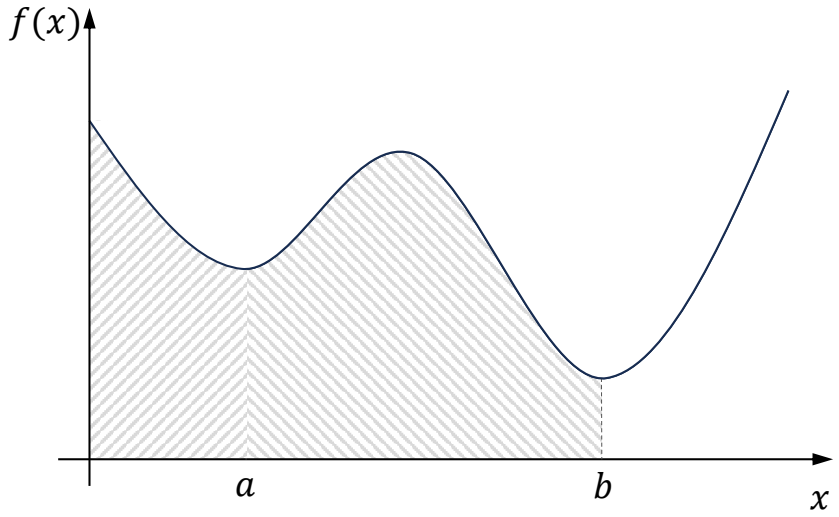
Rozważmy **Podstawowe twierdzenie rachunku całkowego**, które brzmi następująco:

Twierdzenie 2: Podstawowe twierdzenie rachunku całkowego.

$$\int_a^b f(x)\, dx = F(b) - F(a) \tag{1.9}$$

gdzie $F(x)$ oznacza całkę nieoznaczoną w punkcie x. Powyższe twierdzenie mówi o tym, że aby wyznaczyć pole powierzchni pod wykresem funkcji f na przebiegu między a i b, należy obliczyć: pole powierzchni od $-\infty$ do a, pole powierzchni pod funkcją od $-\infty$ do b, i odjąć te dwie wartości. Poniżej intuicja tego wzoru:

Dla uproszczenia oznaczeń, przyjmijmy, że $\int f'(x)dx = f(x)$, wtedy powyższy wzór zmienia się w:



?figurename? 1.2: Całka oznaczona

$$f(b) - f(a) = \int_a^b f'(x)dx$$

Uzyskujemy dalej taką samą intuicję, przy czym trochę uprościliśmy przyjmowaną notację. Idąc o krok dalej, przyjmijmy następujące oznaczenia: niech $a = x$, $b = x + h$. Niech punkt a będzie punktem startowym (porównawczym, referencyjnym), a punkt b punktem uzyskanym po przesunięciu o wartość h względem punktu a. Warto zwrócić uwagę na fakt, że wraz ze wzrostem wartości h zwiększa się również pole powierzchni pod krzywą funkcji f, gdyż rośnie odległości od punktu referencyjnego.

$$f(x + h) - f(x) = \int_x^{x+h} f'(a)da$$

Porządkując powyższe równanie, oraz sprowadzając całkę oznaczoną do początku układu współrzędnych (teraz jest sztucznie zaczepiona w punkcie x – zwróć uwagę, że początek całki jak i jej koniec są funkcjami x), otrzymujemy:

$$f(x + h) = f(x) + \int_0^h f'(x + a)da \tag{1.10}$$

Powyższy wzór 1.10 jest istotny z perspektywy naszych dalszych przekształceń. Można zauważyć w nim, że x jest interpretowany jako stała wartość (patrz: całka jest po a). Co więcej, można zauważyć że wyrażenie po lewej stronie równości wstępuje w zbliżonej formie pod znakiem całki. Spróbujmy zatem wyrażenie pod całką rozwinąć zgodnie ze wzorem który otrzymaliśmy:

$$f(x + h) = f(x) + \int_0^h \left[f'(x) + \int_0^a f''(x + b)db \right] da$$

Korzystając z własności addytywności całki, uzyskujemy:

$$f(x+h) = f(x) + \int_0^h f'(x)da + \int_0^h \left[\int_0^a f''(x+b)db\right] da$$

Spróbujmy rozpisać pierwszą z całek w powyższym wzorze:

$$\int_0^h f'(x)da = [f'(x)a]_0^h = f'(x)h$$

Czyli łącznie uzyskujemy:

$$f(x+h) = f(x) + f'(x)h + \int_0^h \int_0^a f''(x+b)db\, da$$

Wyrażenie pod znakiem podwójnej całki możemy również rozpisać przy pomocy opisanego wcześniej wzoru:

$$f(x+h) = f(x) + f'(x)h + \int_0^h \int_0^a f''(x) + \left[\int_0^b f''(x+c)dc\right] db\, da$$

Spróbujmy teraz rozpisać ponownie, pierwszą całkę we wzorze:

$$\int_0^h \int_0^a f''(x)db\, da = \int_0^h [f''(x)b]_0^a\, da = \int_0^h f''(x)a\, da = \int_0^h \left[\frac{1}{2}f''(x)a^2\right]_0^h = \frac{1}{2}f''(x)h^2$$

Łącząc to wyprowadzenie z naszym wcześniejszym równaniem, otrzymujemy:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \int_0^h \int_0^a \int_0^b f''(x+c)dc\, db\, da$$

Łatwo zauważyć, że podobnie jak wcześniej – wyrażenie w całce jest rozpisywalne przy pomocy naszego wcześniejszego równania 1.10; co więcej, taką procedurę można powtarzać w nieskończoność (technicznie, tyle razy ile razy różniczkowalna jest funkcja $f(x)$ – w powyższym wzorze to łatwo zauważyć). W ogólności wzór ten można sprowadzić do:

$$f(x+h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}h^n$$

Powyższy wzór jest nazywany *równaniem / wzorem Taylora.*

Zwróćmy uwagę, że w praktyce często nie będziemy mogli nieskończenie wiele razy różniczkować f. Nieskończenie długie obliczenia leżą niestety poza naszym zasięgiem. Na szczęscie możemy im zapobiec rozpisując ten wzór dla konkretnego, N-tego rozwinięcia funkcji wzorem Taylora:

$$f(x+h) = f(x) + \sum_{n=1}^{N-1} \frac{1}{n!}f^{(n)}(x)h^n + \frac{f^{(N)}(x+\theta h)}{N!}h^N$$

gdzie $R_N(x,h) = \frac{f^{(N)}(x+\theta h)}{N!}h^N$ jest resztą Lagrange'a. Można również pokazać, że reszta ta ma następującą własność:

$$\lim_{h\to 0} \frac{R_N(x,h)}{h^N} = 0$$

Oznacza to, że reszta $R_N(x,h)$ aproksymacji wzorem Taylora maleje do 0 w tempie szybszym niż wielomian N-tego stopnia.

Twierdzenie 3: Wzór Taylora $f:\mathbb{R}\to\mathbb{R}$**.**
Niech $f:\mathbb{D}\subset\mathbb{R}\to\mathbb{R}$ *oraz* $f\in C^N$ *w każdym punkcie odcinka* $[x,x+h]$*. Wówczas dla pewnego* θ *mamy:*

$$f(x+h)=f(x)+\sum_{n=1}^{N-1}\frac{1}{n!}f^{(n)}(x)h^n+\frac{f^{(N)}(x+\theta h)}{(N)!}h^N$$

Twierdzenie Taylora jest często wykorzystywane do przybliżania funkcji rzeczywistych, więc warto o nim pamiętać.

Rozwinięcie funkcji w szereg Taylora 1 i 2 stopnia:

$$f(x+h)\approx f(x)+f'(x)h+\frac{1}{2}f''(x)h^2$$

Przykład 2. Rozwiń funkcję $f(x)=\sin x$ w szereg Taylora w okolicy punktu $x_0=0$.

Wyznacz pochodne funkcji:

$$\begin{aligned}(\sin x)'&=\cos x\\(\sin x)''&=-\sin x\\(\sin x)'''&=-\cos x\\(\sin x)^{(4)}&=\sin x\end{aligned}$$

Można zauważyć, że wzorzec te powtarza się dla pochodnych wyższego rzedu. Wyznaczym teraz pochodną w punkcie 0.

$$\begin{aligned}\sin 0&=0\\(\sin 0)'&=1\\(\sin 0)''&=0\\(\sin 0)'''&=-1\\(\sin 0)^{(4)}&=0\end{aligned}$$

Używając wzoru Taylora otrzymujemy:

$$\begin{aligned}sin(x)&=0+1x-0x^2+\frac{-1}{3!}x^3+0x^4+\dots\\&=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\dots\end{aligned}$$

Przy wykorzystaniu języka **R** rozwińmy przykładową funkcję w szereg Taylora 1 i 2 stopnia: $f(x)=\frac{x^2}{e^x}$:

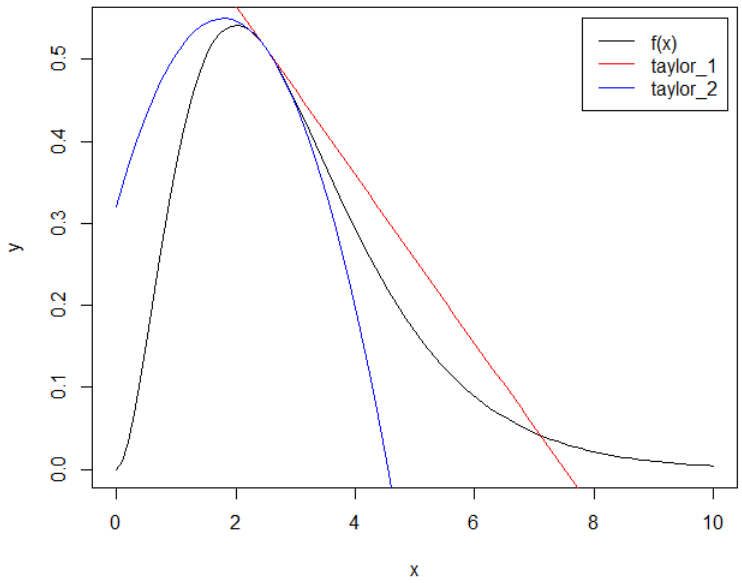
```r
# Dane wejsciowe
f  <- function(x) x^2/exp(x)
x0 <- 2.5
h_seq <- seq(0, 10, length = 100)

# Pochodne numeryczne
d1f <- function(f, x, h = 10^-6) (f(x+h)-f(x))/h
d2f <- function(f, x, h = 10^-6) (f(x+2*h)-2*f(x+h)+f(x))/h^2

# Aproksymacja Taylora funkcji f wokol x0
taylor_1 <- function(f, x, h) f(x)+d1f(f, x)*(h-x)
taylor_2 <- function(f, x, h) f(x)+d1f(f, x)*(h-x)+1/2*d2f(f, x)*(h-x)^2

# Wykresy
plot(h_seq, f(h_seq), type='l', col='black', xlab = 'x', ylab = 'y')
lines(h_seq, taylor_1(f, x0, h_seq), col='red')
lines(h_seq, taylor_2(f, x0, h_seq), col='blue')
legend(7.8, 0.55, legend=c('f(x)', 'taylor_1', 'taylor_2'),
       col=c('black', 'red', 'blue'), lty=1, cex=1)
```

Listing 1: Przykład rozwiniecia funkcji w szereg Taylora



?figurename? 1.3: Przykład: rozwinięcie funkcji w szereg Taylora

1.2.3 Warunki Drugiego Rzędu $f : \mathbb{R} \to \mathbb{R}$

Wskazaliśmy wcześniej, że Warunki Pierwszego Rzędu stanowią warunki konieczne (tj. każde ekstremum będzie posiadać taką własność), ale niewystarczające (tj. są punkty, które ekstremami nie są, a również posiadają taką własność). W celu weryfikacja czy dany punkt stacjonarny jest ekstremum, należy wykorzystać warunki *dostateczne* – Warunki Drugiego Rzędu.

Twierdzenie 4: Warunki Drugiego Rzędu $f : \mathbb{R} \to \mathbb{R}$.
Niech $f : \mathbb{D} \subset \mathbb{R} \to \mathbb{R}$, $f \in C^n$. Jeżeli dla pewnego $x \in \mathbb{D}$ zachodzi: $f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$, ale $f^{(n)}(x) \neq 0$, to:

1. *jeśli n jest parzyste, to funkcja f ma w punkcie x ekstremum; jeśli $f^{(n)}(x) > 0$ to jest to minumum, $f^{(n)}(x) < 0$ to jest to maksimum*
2. *jeśli n jest nieparzyste, to funkcja f ma w punkcie x nie ma ekstremum.*

?proofname? Ze wzoru Taylora dla pewnego $0 < \theta < 1$ mamy:

$$f(x+h) = \sum_{k=0}^{n-1} \frac{1}{k!} f^{(k)}(x) h^k + \frac{1}{(n)!} f^{(n)}(x+\theta h) h^{(n)}$$

a ponieważ $f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$ to:

$$f(x+h) = f(x) + \frac{1}{n!} f^{(n)}(x+\theta h) h^n$$

$$f(x+h) - f(x) = \frac{1}{n!} f^{(n)}(x+\theta h) h^n$$

Kiedy n jest parzyste, i $f^{(n)}(x) > 0$, ze względu na parzystość n, mamy $h^n > 0$. Ze względu na ciągłość funkcji $f^{(n)}$ w punkcie n wiemy, że istnieje $\delta > 0$ taka, że dla każdego $h : 0 < |h| < \delta$ mamy $f^{(n)}(x+h) > 0$, a więc także $f^{(n)}(x+\theta h) > 0$. Oznacza to, że funkcja f ma w tym miejscu minimum. Analogiczne możemy wykazać przypadek maksimum. □

1.3 Optymalizacja nieliniowa bez ograniczeń, przypadek *wielowymiarowy*, tj. $f : \mathbb{R}^n \to \mathbb{R}$

We wcześniejszej części wykazaliśmy warunki optymalizacji dla przypadku jednowymiarowego – tj. sytuacji, w które mamy do czynienia z jedną zmienną decyzyjną. Natura problemów optymalizacyjnych jest z reguły jednak wielowymiarowa. Poniżej przedstawiamy analogiczny wywód w zakresie problemów wielowymiarowych.

Na początek jednak powinniśmy wyprowadzić kilka definicji z obszaru rachunku różniczkowego funkcji wielu zmiennych. Warto w tym przypadku sięgnąć po podręcznik [BI86] – w szczególności początkowe rozdziały.

Definicja 4: Pochodna kierunkowa

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 kierunkową funkcji f w punkcie x w kierunku h nazywamy funkcję:

$$\frac{df}{dh}(x) = \lim_{t \to 0} \frac{f(x+th) - f(x)}{t}$$

Definicja 5: 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 funkcji f w punkcie x względem zmiennej x_i, $i = 1, 2, ...n$ nazywamy 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 f względem x_i jest więc pochodną kierunkową f w kierunku $i$tego wersowa, tj. przy $h = e_i$

Definicja 6: Gradient funkcji

Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. Gradientem funkcji f w punkcie x nazywamy funkcję $\nabla_f(x) : \mathbb{R}^n \to \mathbb{R}^n$:

$$\nabla_f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)\right]$$

Związek między Pochodną Kierunkową a Gradientem? Jeżeli istnieje gradient funkcji $\nabla_f(\mathbf{x})$ w punkcie $\mathbf{x}$ (co oznacza, że f jest różniczkowalna w $\mathbf{x}$)

$$\nabla_f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

to pochodna kierunkowa funkcji f w kierunku wektora $\mathbf{h}$ jest równa iloczynowi skalarnemu gradientu funkcji f i wektora $\mathbf{h}$

$$\frac{df}{dh} = \nabla_f(\mathbf{x}|\mathbf{h}) = \nabla_f(\mathbf{x}) \times \mathbf{h}$$

1.3.1 Warunki Pierwszego Rzędu $f : \mathbb{R}^n \to \mathbb{R}$

Twierdzenie 5: Warunki Pierwszego Rzędu $f : \mathbb{R}^n \to \mathbb{R}$.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$. Jeżeli funkcja f posiada ekstremum w punkcie x, to $\nabla_f(x) = \mathbf{0}$

?proofname? Rozważmy funkcję $g_x(t) = f(x + th)$ oraz $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Ponieważ f ma ekstremum w x, to g ma ekstremum w $t = 0$. Zatem $g'(t) = 0$ dla $t = 0$, co będzie oznaczane jako $g'(t)|_{t=0} = 0$. W konsekwencji:

$$g'(t)\Big|_{t=0} = \lim_{\Delta \to 0} \frac{g(t + \Delta) - g(t)}{\Delta}\Big|_{t=0} = \lim_{\Delta \to 0} \frac{f(x + \Delta h) - f(x)}{\Delta} = \frac{df}{dh}(x) = \nabla_f(x)h = \mathbf{0}$$

□

Przykład 3. Przykład. Niech $f(x) = x_1^2 + x_2^2$. Znajdź ekstrema funkcji $f(x)$.

$$\nabla_f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x)\right] = [2x_1, 2x_2] = [0, 0]$$

1.3.2 Warunki Drugiego Rzędu $f : \mathbb{R}^n \to \mathbb{R}$

Do wyprowadzenia warunków drugiego rzędu potrzebujemy wskazać na uogólnienie pochodnej drugiego rzędu dla przypadku funkcji wielu zmiennych: tzw. macierz Hessego.

Definicja 7: Macierz Hessego

Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. Macierzą Hessego $H_f(x)$ nazywamy macierz:

$$H_f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \dots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{bmatrix}$$

Uwaga! Macierz Hessego, jest macierzą symetryczną wtedy, gdy wszystkie pochodne cząstkowe drugiego stopnia są ciągłe (tzw. twierdzenie Schwarza). Oznacza to, że: $\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{\partial^2 f}{\partial x_j \partial x_i}(x)$

Dodatkowo, wcześniej wyprowadziliśmy wzór Taylora dla przypadku jednowymiarowego; poniżej jest wzór Taylora w przypadku wielowymiarowym.

Twierdzenie 6: Wzór Taylora $f : \mathbb{R}^n \to \mathbb{R}$.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$ oraz $f \in C^2$ w każdym punkcie odcinka $[x, x + h]$. Wówczas dla mamy:

$$f(x + h) = f(x) + \nabla_f(x)h + \frac{1}{2}h^T H_f(x)h + R_3(x, h)$$

Twierdzenie 7: Warunki Drugiego Rzędu $f : \mathbb{R}^n \to \mathbb{R}$.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^2$. Jeżeli w x^* zachodzi

1. $\nabla_f(x^*) = \mathbf{0}$, *oraz*
2. $H_f(x^*) > 0$

Wtedy w x^* jest minimum lokalne f.

?proofname? Z twierdzenia Taylora dla $\mathbb{R}^n$ mamy:

$$f(x+h) = f(x) + \nabla_f(x)h + \frac{1}{2}h^T H_f(x)h + o(|h|^2) = f(x) + \frac{1}{2}h^T H_f(x)h + o(|h|^2)$$

czyli $f(x+h)-f(x) = \frac{1}{2}h^T H_f(x)h+o(|h|^2)$. Z twierdzenia Rayleigh'a, wartość formy kwadratowej $h^T H_f(x)h$ możemy ograniczyć z dołu przez $\lambda_{min}|h|^2$:

$$f(x+h) - f(x) = \frac{1}{2}h^T H_f(x)h + o(|h|^2) \geqslant \frac{1}{2}\lambda_{min}|h|^2 + o(|h|^2)$$

Dla dostatecznie małych h mamy więc $f(x+h) - f(x) > 0$ □

To co nam jeszcze pozostaje, to sprawdzić kiedy $H_f(x^*) > 0$, tj. macierz $H_f(x^*)$ jest macierzą dodatnio określoną.

Twierdzenie 8: Kryterium Sylwestera.
Niech A będzie symetryczną macierzą o wartościach rzeczywistych:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}$$

Minorami głównymi macierzy nazwiemy:

$$M_1 = a_{1,1} \quad M_2 = det\left(\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}\right) \quad \dots \quad M_n = det\left(\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}\right)$$

Wówczas:

1. *Macierz jest dodatnio określona wtedy i tylko wtedy, gdy **wszystkie** minory główne M_i macierzy A są dodatnie.*
2. *Macierz jest ujemnie określona wtedy i tylko wtedy, gdy wszystkie **parzyste** minory główne M_i macierzy A są dodatnie, a wszystkie **nieparzyste** minory główne M_i są ujemne.*

Przykład 4. Niech $f(x) = x_1^2 + x_2^2$. Znajdź ekstrema funkcji $f(x)$:

Twierdzenie Warunków Pierwszego Rzędu:

$$\nabla_f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x)\right] = [2x_1, 2x_2] = [0, 0]$$

Twierdzenie Warunków Drugiego Rzędu:

$$H_f([0,0]) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2^2}(x) \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} > 0$$

Funkcja $f(x)$ posiada jedno ekstremum (minimum) w punkcie $[0,0]$.

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 2: Optymalizacja analityczna z ograniczeniami

Daniel Kaszyński

2.4 Własności gradientu funkcji

Pojęciem już wprowadzonym jest pojęciu gradientu funkcji – wektora pochodnych cząstkowych. Gradient funkcji będzie często wykorzystywany w ramach wykładu, stąd warto znać jego dwie, istotne z perspektywy optymalizacji, własności. Dla przypomnienia:

Definicja 8: Gradient funkcji

Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. Gradientem funkcji f, jest funkcja $\nabla_f(x) : \mathbb{R}^n \to \mathbb{R}^n$ w punkcie x nazywamy funkcję:

$$\nabla_f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)\right]$$

Pamiętajmy, że:

Związek między Pochodną Kierunkową a Gradientem? Jeżeli istnieje gradient funkcji $\nabla_f(\mathbf{x})$ w punkcie $\mathbf{x}$ (co oznacza, że f jest różniczkowalna w $\mathbf{x}$)

$$\nabla_f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

to pochodna kierunkowa funkcji f w kierunku wektora $\mathbf{h}$ jest równa iloczynowi skalarnemu gradientu funkcji f i wektora $\mathbf{h}$

Twierdzenie 9: Gradient jest kierunkiem najszybszego wzrostu wartości funkcji.

Proof. Niech $|h| = 1$, tj. będzie znormalizowanym wektorem. Wtedy stopa wzrostu wartości funkcji f w punkcie x w kierunku h jest dana przez pochodną kierunkową $\frac{df}{dh}(x)$. Wyznaczmy w takim razie kierunek h, który maksymalizuje stopę wzrostu wartości funkcji f, tj. kierunek maksymalizujący pochodną kierunkową:

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

dla $|\nabla_f(x)| \geqslant 0$ oraz $cos(\nabla_f(x), h) \in [-1, 1]$ stopa wzrostu wartości funkcji f jest największa, gdy $cos(\nabla_f(x), h) = 1$, co implikuje, że h wskazuje ten sam kierunek co $\nabla_f(x)$. Oznacza to, że $h = \frac{\nabla_f(x)}{|\nabla_f(x)|}$. □

Twierdzenie 10: Gradient jest ortogonalny względem warstwicy funkcji.

Proof. Niech $f : \mathbb{R}^n \to \mathbb{R}$, $x^* = (x_1^*, \ldots, x_n^*)$ oraz $\nabla_f(x^*) \neq 0$. Niech $r : \mathbb{R} \to \mathbb{R}^n$ taki, że $r(t_0) = x^*$. Wartość funkcji jest stała dla wszystkich punktów z zadanej warstwicy (zgodnie z definicją warstwicy) oraz: $\forall_{t\in\mathbb{R}} f(r(t)) = c$. Wtedy $\frac{d}{dt}(f(r(t)) = \nabla_f(r(t))\frac{dr}{dt}(t) = 0$. W szczególności $\nabla_f(r(t_0))\frac{dr}{dt}(t_0) = 0$. Ponieważ $\frac{dr}{dt}(t_0)$ jest przestrzenią styczną do warstwicy funkcji f w x^*, oznacza to wtedy, że $\nabla_f(x^*)$ jest prostopadła do warstwicy funkcji. □

2.4.1 Warunki Pierwszego Rzędu w przypadku ograniczeń równościowych

Twierdzenie 11: Twierdzenie Lagrange'a, Metoda mnożników Lagrange'a.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$. Jeżeli funkcja f posiada w x ekstremum pod warunkiem $h(x) = 0$, gdzie $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^1$, w punkcie x, to

$$\nabla_f(x) + \lambda^T \mathbf{D}h(x) = \mathbf{0}$$

Dla $h : \mathbb{R}^n \to \mathbb{R}$ (w przypadku gdy jest jedno ograniczenie).

$$\nabla_f(x) + \lambda \nabla_h(x) = \mathbf{0}$$

Przykład 5. Niech $f(x) = x_1^2 + x_2^2$, oraz $h(x) = x_1^2 + 2x_2^2$. Znajdź ekstrema funkcji $f(x)$ p.w. $h(x) = 1$. Z Warunków Pierwszego Rzędu (FOC) uzyskujemy:

$$\begin{cases} 2x_1 + \lambda 2x_1 = 0 \Rightarrow x_1(1+\lambda) = 0 \\ 2x_2 + \lambda 4x_2 = 0 \Rightarrow x_2(1+2\lambda) = 0 \end{cases}$$

Co daje nam 4 rozwiązania:

1. $[x_1, x_2] = \left[0, \frac{1}{\sqrt{2}}\right]$, $\lambda = -\frac{1}{2}$
2. $[x_1, x_2] = \left[0, -\frac{1}{\sqrt{2}}\right]$, $\lambda = -\frac{1}{2}$
3. $[x_1, x_2] = [1, 0]$, $\lambda = -1$
4. $[x_1, x_2] = [-1, 0]$, $\lambda = -1$

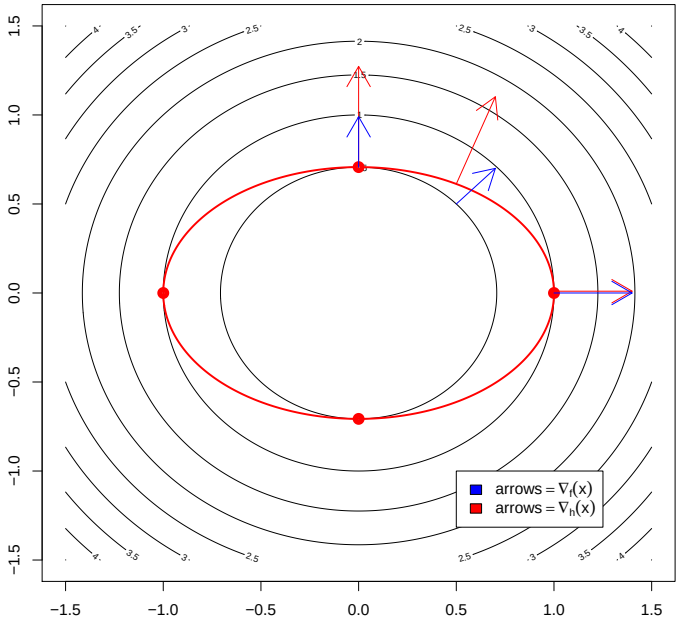


Figure 2.4: Wizualizacja ograniczeń równości Warunków Pierwszego Rzędu

2.4.2 Warunki Drugiego Rzędu w przypadku ograniczeń równościowych

Twierdzenie 12: Warunki Drugiego Rzędu Twierdzenia Lagrange'a.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^2$. *Niech istnieją takie* x *oraz* λ, *że:*

1. $\nabla_f(x) + \lambda^T \mathbf{D}h(x) = 0$, *oraz*
2. $\forall_{z \in T(x), z \neq 0}$ *mamy* $z^T H_{L(x)} z > 0$

Wtedy punkt x *nazywamy minimum funkcji* f, *pod warunkiem* $h(x) = 0$. $T(x)$ *nazywamy przestrzenią styczną, tj:* $T(x) = \{z \in \mathbb{R}^n : z^T \mathbf{D}h(x) = 0\}$. *W przypadku, gdy* $m = 1$, *mamy* $T(x) = \{z \in \mathbb{R}^n : z^T \nabla_h(x) = 0\}$

Przykład 6. Rozważmy 4 rozwiązania, które uzyskaliśmy z FOC.

1. $[x_1, x_2] = \left[0, \frac{1}{\sqrt{2}}\right]$, $\lambda = -\frac{1}{2}$

$$z : z^T \nabla_h(x) = [z_1, z_2][2x_1, 4x_2]^T = [z_1, z_2] \left[0, \frac{4}{\sqrt{2}}\right] = 0$$

$$z_1 0 + z_2 \frac{4}{\sqrt{2}} = 0 \Rightarrow \mathbf{z} = [\alpha, 0]$$

$$[\alpha,0]^T H_f([x_1,x_2])[\alpha,0] = [\alpha,0]^T \begin{bmatrix} 2+2\lambda & 0 \\ 0 & 2+4\lambda \end{bmatrix} [\alpha,0] = [\alpha,0]^T \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} [\alpha,0] = \alpha^2 > 0$$

2. $[x_1,x_2] = \left[0, -\frac{1}{\sqrt{2}}\right]$, $\lambda = -\frac{1}{2}$
3. $[x_1,x_2] = [1,0]$, $\lambda = -1$

$$z : z^T \nabla_h(x) = [z_1,z_2][2x_1,4x_2]^T = [z_1,z_2]\,[1,0] = 0$$

$$z_1 1 + z_2 0 = 0 \Rightarrow \mathbf{z} = [0,\alpha]$$

$$[0,\alpha]^T H_f([x_1,x_2])[0,\alpha] = [0,\alpha]^T \begin{bmatrix} 2+2\lambda & 0 \\ 0 & 4+\lambda \end{bmatrix} [0,\alpha] = [0,\alpha]^T \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix} [0,\alpha] = -2\alpha^2 < 0$$

4. $[x_1,x_2] = [-1,0]$, $\lambda = -1$

2.4.3 Warunki Pierwszego Rzędu w przypadku ograniczeń *nie*równościowych

Twierdzenie 13: Warunki Pierwszego rzedu Karush-Kuhn-Tucker, KKT.
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$ będzie funkcją celu, oraz $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^1$ oraz $g : \mathbb{R}^n \to \mathbb{R}^p$, $g \in C^1$ będą ograniczeniami. Jeżeli x jest ekstremum, to istnieje taki zestaw $\lambda = (\lambda_1,\ldots,\lambda_n)$ oraz $\mu = (\mu_1,\ldots,\mu_n)$ że:

1. ***Stationarity condition**:* $\nabla_f(x) + \sum_{i=1}^m \lambda_i \nabla_{h_i(x)} + \sum_{i=1}^p \mu_i \nabla_{g_i(x)} = 0$
2. ***Primal feasibility**:* $\forall_{i=1,\ldots,m}\ h_i(x) = 0$ *and* $\forall_{i=1,\ldots,p} g_i(x) \leqslant 0$
3. ***Dual feasibility**:* $\forall_{i=1,\ldots,p}\ \mu_i \geqslant 0 \leftarrow$ *dla minimum*
4. ***Complementary slackness**:* $\forall_{i=1,\ldots,p}\ \mu_i g_i(x) = 0$

Przykład 7. $f(x) = x_1^2 + x_2^2$ ograniczona przez $[x_1,x_2] : g(x) = x_1^2 + 2x_2^2 - 1 \leqslant 0$

Z FOC mamy:

$$[2x_1, 2x_2] + \mu[2x_1, 4x_2] = 0$$

$$\begin{cases} 2x_1 + \mu 2x_1 = 0 \Rightarrow x_1(1+\lambda) = 0 \\ 2x_2 + \mu 4x_2 = 0 \Rightarrow x_2(1+2\lambda) = 0 \end{cases}$$

Co daje nam 5 rozwiązań:

1. $[x_1,x_2] = \left[0, \frac{1}{\sqrt{2}}\right]$, $\mu = -\frac{1}{2}$ **Dual feasibility**
2. $[x_1,x_2] = \left[0, -\frac{1}{\sqrt{2}}\right]$, $\mu = -\frac{1}{2}$ **Dual feasibility**

3. $[x_1, x_2] = [1, 0]$, $\mu = -1$ **Dual feasibility**

4. $[x_1, x_2] = [-1, 0]$, $\mu = -1$ **Dual feasibility**

5. $[x_1, x_2] = [0, 0]$, $\mu = 0$ **Stationarity condition**

Rozwiązania od 1-4 posiadają wartości $\mu < 0$. Nie spełniają one zatem warunku *Dual feasibility*. Jedynym rozwiązaniem, które je spełnia jest rozwiązanie numer 5.

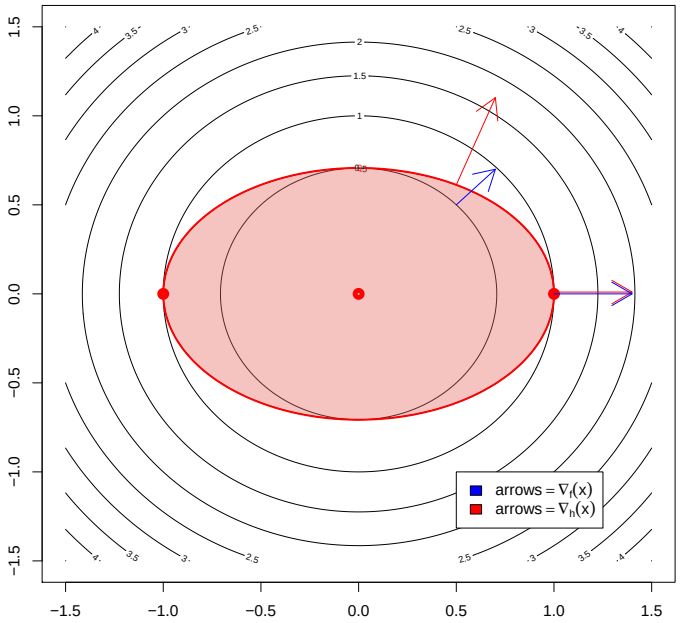


Figure 2.5: Wizualizacja ograniczeń nierówności Warunków Pierwszego Rzędu

2.4.4 Warunki Drugiego Rzędu w przypadku ograniczeń nierównościowych

Twierdzenie 14: Twierdzenie SOC (Twierdzenie KKT).
Niech $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^2$, $g : \mathbb{R}^n \to \mathbb{R}^p$, $g \in C^2$. *Niech istnieją takie* x, λ *oraz* μ, *że: 1.* $\nabla_f(x) + \lambda^T \mathbf{D}h(x) + \mu^T \mathbf{D}g(x) = 0$, *oraz 2.* $\forall_{z \in T(x), z \neq 0}$ *mamy* $z^T H_{L(x)} z > 0$

Wtedy punkt x *nazywamy minimum funkcji* f, *związane* $h(x) = 0$.

$T(x)$ *nazywamy przestrzenią styczną, tj:* $T(x) = \{z \in \mathbb{R}^n : z^T \mathbf{D}h(x) = 0\}$. *W przypadku, gdy* $m = 1$, *mamy* $T(x) = \{z \in \mathbb{R}^n : z^T \nabla_h(x) = 0\}$

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 3: Aproksymacje numeryczne

Daniel Kaszyński

3.5 Różnice skończone

Różnice skończone to wyrażenia matematyczne w postaci $f(x+b) - f(x+a)$. Jeśli podzielimy różnicę skończoną przez $b-a$, otrzymamy iloraz różnicowy. Są one często stosowane w metodach różnic skończonych do numerycznego rozwiązywania równań różniczkowych.

3.5.1 Różnice skończone wstecz i w przód

Najpopularniejszymi metodami aproksymacji pochodnych są różnice skończone wstecz i w przód.

Definicja 9: Pochodna funkcji

Pochodną funkcji $f : \mathbb{R} \to \mathbb{R}$ opisaną przez **różnicę skończoną w przód** nazywamy:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.11}$$

Pochodną funkcji $f : \mathbb{R} \to \mathbb{R}$ opisaną przez **różnicę skończoną wstecz** nazywamy:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{3.12}$$

Algorytmy różniczkowania numerycznego szacujące pochodną funkcji matematycznej przy użyciu różnicy skończonej w przód lub różnicy skończonej wstecz obarczone są błędem $O(h)$ (można to udowodnić przy pomocy twierdzenia Taylora).

Z twierdzenia Taylora wiemy, że:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{3.13}$$

Wyrażenie to możemy przekształcić na:

$$f'(x)h = f(x+h) - f(x) - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 - ... \tag{3.14}$$

i dzieląc to wyrażenie przez h otrzymujemy:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{3.15}$$

Możemy więc wywnioskować, że błąd różnicy w przód jest rzędu $O(h)$.

Stosując tę samą metodologię, możemy przekształcić poniższy wzór:

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{3.16}$$

we wzór na różnicę wstecz:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{3.17}$$

Różnica wsteczna ma również błąd rzędu $O(h)$.

3.5.2 Centralna różnica skończona

Oprócz wczesniejszych dwóch sposobów przybliżania pochodnej funkcji, można wskazać również trzeci, wykorzystujący **różnicę centralną**. Ta metoda jest czasami nazywana pochodną symetryczną.

Definicja 10: Centralna różnica skończona funkcji

Pochodną funkcji $f : \mathbb{R} \to \mathbb{R}$ opisaną przez **centralną różnicę skończoną** nazywamy:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{3.18}$$

Algorytmy różniczkowania numerycznego szacujące pochodną funkcji matematycznej za pomocą różnicy centralnej obarczone są błędem $O(h^2)$. Jest to preferowane, gdyż błąd jest wtedy mniejszy niż w przypadku poprzednich metod (pamiętajmy, że zakładamy iż $h \to 0$).

Podobnie jak w przypadku opisanych wcześniej przybliżeń, z twierdzenia Taylora możemy wyprowadzić wzór na różnicę centralną. Tym razem jednak będziemy potrzebować dwóch równań wyjściowych:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{3.19}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{3.20}$$

Odejmując od siebie oba powyższe wzory, otrzymujemy:

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{1}{3}f'''(x)h^3 + ... \tag{3.21}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}f'''(x)h^3 - ... \tag{3.22}$$

3.6 Błąd anulowania subtraktywnego

Obliczając pochodne funkcji, możemy napotkać problemy nie tylko natury matematycznej (tj. przejście z rachunku różniczkowego na rachunek różnic skończonych), ale także techniczne/sprzętowe. Jednym z popularnych problemów tego rodzaju jest **błąd anulowania subtraktywnego**.

Podczas stosowania arytmetyki zmiennoprzecinkowej możemy napotkać błąd anulowania subtraktywnego. Kiedy komputery pracują z liczbami rzeczywistymi, muszą w jakiś sposób przechowywać w tych liczbach

potencjalnie nieskończoną liczbę miejsc po przecinku. W tym celu wykorzystują m.in. zmienne typu float, które stanowią skończone przybliżenie liczb rzeczywistych. Większość języków programowania wykorzystuje standard techniczny arytmetyki zmiennoprzecinkowej zwany **IEEE 754**. Niestety, w ten sposób częściowo traci się precyzję, ale jest to spowodowane ograniczoną pamięcią komputera.

Błąd anulowania subtraktywnego to zjawisko, które może wystąpić podczas odejmowania dwóch prawie równych liczb. Liczby zmiennoprzecinkowe w komputerach mają ograniczoną precyzję i gdy odejmie się dwie liczby o bardzo zbliżonej wartości, w wyniku może wystąpić utrata cyfr znaczących.

Przykład 8. Niech $a = 0.3 + 0.3 + 0.4 - 1$ oraz $b = -1 + 0.3 + 0.3 + 0.4$.

Kierując się prostą matematyką, możemy stwierdzić, że a jest równe b. Niestety obliczenia wykonane na zmiennych zmiennoprzecinkowych mogą nie dać tego samego wyniku.

```r
a <- 0.3+0.3+0.4-1
b <- -1+0.3+0.3+0.4

print(a == b) # FALSE
print(a) # 0
print(b) # 5.551115e-17
```

Listing 2: Przykład błędu anulowania subtraktywnego w języku **R**

W tym przykładzie obliczenia wykonane w celu uzyskania a i b mogą dać bardzo zbliżone, ale jednak różne wartości. Wyniki mogą nie być na tyle dokładne, na ile można by się spodziewać, ze względu na błąd anulowania subtraktywnego. Dokładność wyniku zależy od liczby cyfr znaczących, które można przedstawić w formacie zmiennoprzecinkowym. Trzeba zatem pamiętać, że przy odejmowaniu zmiennych typu float może wystąpić błąd, który choć niewielki, może zepsuć niektóre proste porównania i obliczenia.

Aby złagodzić błędy anulowania subtraktywnego, można zastosować różne techniki i algorytmy, takie jak zmiana układu wyrażenia, aby uniknąć odejmowania prawie równych liczb lub w razie potrzeby użyć arytmetyki o większej precyzji. Ponadto zrozumienie ograniczeń arytmetyki zmiennoprzecinkowej i świadomość potencjalnych źródeł błędów jest kluczowa podczas pracy z obliczeniami numerycznymi w programach komputerowych.

3.7 Złożona pochodna schodkowa

Aby uniknąć problemu błędu anulowania subtraktywnego omówionego w poprzedniej sekcji, można zastosować różne metody oraz transformacje wzorów. Jednym z możliwych rozwiązań jest użycie **złożonej pochodnej schodkowej** (ang. *Complex Step Derivative*). Główną ideą tej metody jest wykorzystanie równania Taylora i liczb zespolonych w celu wyeliminowania konieczności odejmowania dwóch zmiennych zmiennoprzecinkowych.

Zacznijmy od równania Taylora używając liczb zespolonych:

$$f(x + ih) = f(x) + f'(x)ih - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)ih^3 + ... \tag{3.23}$$

Zakładając, że chcemy obliczyć pierwszą pochodną funkcji, musimy przekształcić wzór:

$$f'(x)ih = f(x + ih) - f(x) + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)ih^3 + ... \tag{3.24}$$

Następnie musimy wyizolować pochodną. Możemy zacząć od podzielenia obu stron przez h:

$$f'(x)i = \frac{f(x+ih) - f(x)}{h} + \frac{1}{2}f''(x)h + \frac{1}{6}f'''(x)ih^2 + ... \tag{3.25}$$

Aby otrzymać tylko pochodną, musimy zadbać także o część urojoną:

$$f'(x) = Im\left(\frac{f(x+ih) - f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.26}$$

Na szczęście jesteśmy w stanie usunąć $\frac{1}{2}f''(x)h$, ponieważ nie zawiera części urojonej. Uprości to naszą formułę, ale nadal nie jest tak dobra, jak byśmy sobie tego życzyli. Nie pozbyliśmy się jeszcze odejmowania dwóch instancji funkcji. W tym celu powinniśmy rozdzielić pierwszą część naszego wzoru:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) - Im\left(\frac{f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.27}$$

Możemy zauważyć, że $Im\left(\frac{f(x)}{h}\right)$ nie ma części urojonej. Możemy więc usunąć go z naszego równania. W ten sposób otrzymujemy wzór, który pomaga uniknąć problemu błędu anulowania subtraktywnego:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.28}$$

Ten wzór jest esencją złożonej pochodnej schodkowej. Jest to czwarty już wspomniany sposób obliczania pochodnej funkcji. Metoda ta jest obarczona błędem $O(h^2)$ i nie wymaga odejmowania od siebie dwóch instancji funkcji f.

3.8 Porównanie różnic skończonych

Aby lepiej zrozumieć różnice pomiędzy opisanymi różnicami skończonymi, warto przeprowadzić test dla prostej funkcji. Niech $f = sin(x^2)$. Korzystając z zasad różniczkowania symbolicznego, możemy wnioskować, że $f'(x) = 2xcos(x^2)$.

Niech $u = x^2$. Wtedy, $\frac{du}{dx} = 2x$ oraz $\frac{df}{du} = cos(u) = cos(x^2)$. Jeśli połączymy te informacje, otrzymamy następujące równanie:

$$f'(x) = \frac{df}{dx} = \frac{du}{dx}\frac{df}{du} = 2x\cos(x^2) \tag{3.29}$$

Dla pewności możemy użyć języka programowania **R** do obliczenia pochodnej funkcji f. W tym celu musimy najpierw zaimportować bibliotekę `Deriv`, która służy do obliczania pochodnych:

```
1 if(!require(Deriv)) install.packages('Deriv');
```

Listing 3: Importowanie biblioteki `Deriv`

Korzystając z bibliotek zewnętrznych, warto jest sprawdzać ich dokumentację. W języku programowania **R** można to zrobić dodając znak '?' przed nazwą biblioteki:

```
1 # Dostep do dokumentacji biblioteki
2 ?Deriv
```

Listing 4: Dostep do dokumentacji biblioteki `Deriv`

Następnie korzystając z tej biblioteki możemy obliczyć pochodną funkcji f:

```
f <- function(x) sin(x^2);
df <- Deriv(f)

cat('f = ', deparse(f)[2], '\n')
cat('df = ', deparse(df)[2], '\n')
```

Listing 5: Pochodna funkcji f obliczona przy pomocy biblioteki `Deriv`

W wyniku tych obliczeń otrzymujemy:

```
f =  sin(x^2)
df =  2x cos(x^2)
```

Wykresy funkcji f i jej pochodnej df wyglądają następująco:

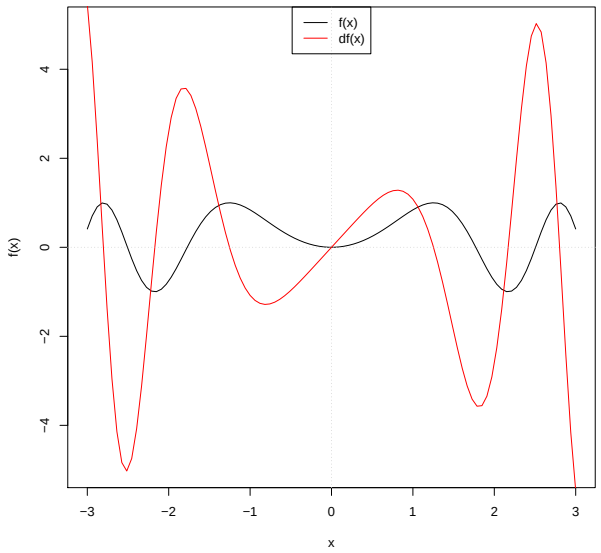


Figure 3.6: Wykresy funkcji f i jej pochodnej df

Oczywiście, aby porównać różne rodzaje aproksymacji pochodnych funkcji w języku programowania **R**, najpierw musimy posiadać implementacje tych metod. Znając definicje i wzory metod, stworzenie ich implementacji nie jest trudnym zadaniem. Przykładowe definicje mogą w kodzie wyglądać następująco:

```
diff_forward <- function (f, x, h = 10^-6) (f(x + h) - f(x)) / (h);
diff_backward <- function (f, x, h = 10^-6) (f(x) - f(x - h)) / (h);
diff_central <- function (f, x, h = 10^-6) (f(x + h) - f(x - h)) / (2*h);
diff_complex <- function (f, x, h = 10^-6 ) Im(f(x + h*1i )) / (h);
```

Listing 6: Implementacje różnic skończonych

Następnie możemy sprawdzić różnice w wartościach otrzymanych w wyniku tych przybliżeń w podanym punkcie $x_0 = 1$. Poniższy fragment kodu zapewnia nam wartości dla każdej skończonej różnicy:

```
x0 <- 1

```

```
cat('df = ' , format( df(x0), nsmall = 20 ), " \n ")
cat('-------------------------------------' , " \n ")
cat('diff_forward = ' , format( diff_forward(f, x0), nsmall = 20), " \n ")
cat('diff_backward = ' , format( diff_backward(f, x0), nsmall = 20), " \n ")
cat('diff_central = ' , format( diff_central(f, x0), nsmall = 20), " \n ")
cat('diff_complex = ' , format( diff_complex(f, x0), nsmall = 20), " \n ")
```

Listing 7: Wartości uzyskane przez różnice skończone dla punktu $x_0 = 1$

```
df             = 1.08060461173627953002
-------------------------------------
diff_forward   = 1.08060346903915416306
diff_backward  = 1.08060575443325035394
diff_central   = 1.08060461179171340973
diff_complex   = 1.08060461173868294082
```

Jak widać wartości te są do siebie zbliżone i nie odbiegają od rzeczywistej wartości pochodnej funkcji. Metoda `diff_complex` osiągnęła najlepszy wynik, ponieważ jest najbliższa wartości prawdziwej pochodej.

Oprócz sprawdzenia poszczególnych wartości funkcji dla każdej ze skończonych różnic, możemy również wykonać wiele innych porównań. Przykładowo możemy wykreślić jak szybko błąd względny zbiega się do 0 w odniesieniu do wszystkich różnic numerycznych. Wykres stworzony w tym celu, powinien mieć skalę wykładniczą, ponieważ wszystko co nas interesuje będzie miało miejsce w bardzo wąskich przedziałach. Aby poprawić widoczność, zaleca się stosowanie skali logarytmicznej.

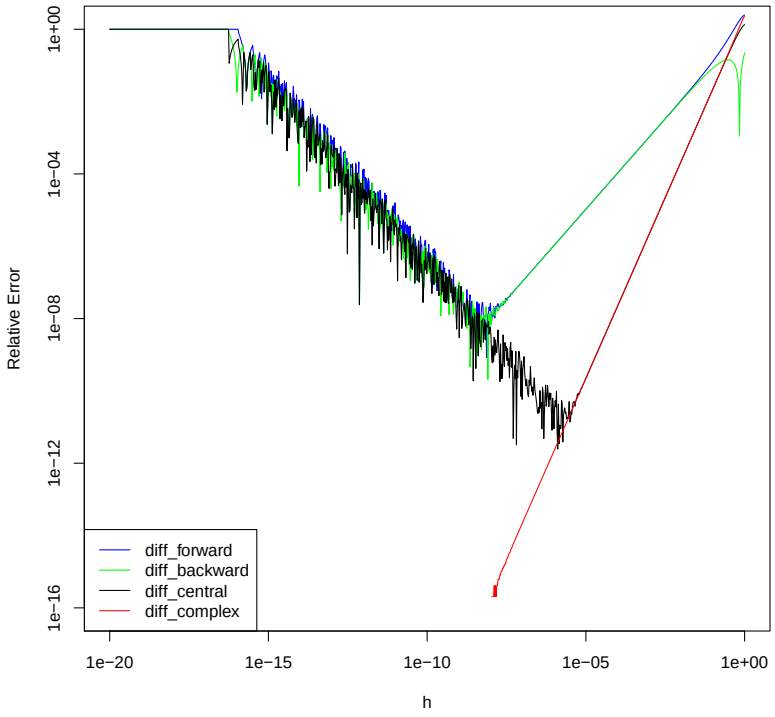


Figure 3.7: Błąd względny dla każdej ze skończonych różnic na f

```r
# Definicja osi x
h <- exp(log(10) * seq (-20, 0, length.out = 1000))

# Wykres bledu wzglednego dla roznicy 'w przod'
plot(h, abs((df(x0) - diff_forward(f, x0, h)) / df(x0)), col = 'blue',
    log = 'xy', type = 'l', ylab = 'Relative Error', ylim = c(10 ^- 16, 1))

# Dodanie do wykresu roznicy 'wstecz', centralnej oraz zlozonej pochodnej schodkowej
lines(h, abs((df(x0) - diff_backward(f, x0, h)) / df(x0)), col = 'green')
lines(h, abs((df(x0) - diff_central(f, x0, h)) / df(x0)), col = 'black')
lines(h, abs((df(x0) - diff_complex(f, x0, h)) / df(x0)), col = 'red')

# Dodanie legendy do wykresu
legend('bottomleft', c("diff_forward", "diff_backward", "diff_central", "diff_complex"),
    col = c("blue" , "green" , "black" , "red"), lty = 1)
```

Listing 8: Implementacja wykresu błędu względnego

Jak widać na powyższym wykresie, charakterystyka zmian błędu względnego w zależności od h jest różna dla każdej metody aproksymacji. Śledząc wartości względem osi h od prawej do lewej, możemy zobaczyć, jak błąd względny zbiega się do zera (lub w większości przypadków przynajmniej próbuje się zbiegać). Dla każdej skończonej różnicy możemy zdefiniować następujące zachowanie:

- **diff_forward** oraz **diff_backward** - Różnica skończona w przód i różnica skończona wstecz zachowywały się bardzo podobnie przy próbie zbiegania do zera. Obie metody osiągnęły minimalny błąd przy h wynoszącym około $1e-08$. Od tego momentu błąd zaczął jednak rosnąć. Było to spowodowane opisanym wcześniej błędem anulowania subtraktywnego (patrz sekcja 3.6). Błąd spowodowany odjęciem dwóch zmiennych zmiennoprzecinkowych dla mniejszych wartości h znacząco wpływał na jakość wyników w tych przypadkach.
- **diff_central** - Centralna różnica zachowywała się podobnie do dwóch wcześniej opisanych metod, z kilkoma zauważalnymi różnicami. Po pierwsze, na początku metoda ta osiągała minimum znacznie szybciej w okolicach $1e-05$. Nachylenie zbiegania błędu względnego dla tego przybliżenia było znacznie bardziej strome. Było to spowodowane faktem, że różnica środkowa obarczona jest błędem $O(h^2)$. Co więcej, wartości błędów względnych pozostawały średnio nieco poniżej wartości uzyskiwanych w przypadku różnic skończonych w przód i wstecz.
- **diff_complex** - Ze wszystkich opisanych metod najlepiej wypadła aproksymacja złożonej pochodnej schodkowej. Nie tylko zbiegała do zera w tempie porównywalnym z różnicą centralną, ale także nie była dotknięta problemem błędu anulowania subtraktywnego. Pozwoliło to na osiągnięcie w tej metodzie wartości błędów względnych bliskich zeru w takim stopniu, że język **R** nie rozróżniał ich od zera (dlatego zniknęły z wykresu).

3.9 Automatyczne różniczkowanie

Automatyczne różniczkowanie (ang. *automatic differentiation*), znane również jako różniczkowanie algorytmiczne lub autoróżnicowanie, to technika stosowana do wydajnego i dokładnego szacowania pochodnych funkcji matematycznych. Podstawowym celem tej techniki jest automatyczne obliczanie pochodnych danej funkcji, co czyni ją szczególnie przydatną w optymalizacji, uczeniu maszynowym i obliczeniach naukowych.

Automatic differentiation różni się od różniczkowania symbolicznego i numerycznego. Różniczkowanie symboliczne napotyka wyzwania podczas przekształcania programu komputerowego w ujednolicone wyrażenie matematyczne, co często skutkuje nieefektywnym kodem. Z drugiej strony różniczkowanie numeryczne, wykorzystujące metodę różnic skończonych, która może wprowadzać błędy w procesie dyskretyzacji. Te tradycyjne

metody sprawiają także problemy przy obliczaniu pochodnych wyższego stopnia. Natomiast automatyczne różnicowanie skutecznie rozwiązuje wszystkie te problemy.

Aby w pełni zrozumieć, jak działa automatyczne różniczkowanie, musimy najpierw zapoznać się z kilkoma podstawowymi koncepcjami, które za nim stoją. Po pierwsze, rozkład różniczek zapewniony przez **regułę łańcuchową** pochodnych cząstkowych ma fundamentalne znaczenie dla automatycznego różniczkowania.

Reguła łańcuchowa to pojęcie w rachunku różniczkowym, które opisuje jak znaleźć pochodną funkcji złożonej. Jeśli mamy złożenie dwóch funkcji $f(x)$ i $g(x)$ takie, że $f(g(x))$, to reguła łańcuchowa pozwala zbadać czy pochodna tego złożenia po x jest iloczynem pochodnej f względem jej argumentu $g(x)$ i pochodnej g względem x:

$$\frac{df}{dx}f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx} = f'(g(x))g'(x) \tag{3.30}$$

Kolejną kwestią, na którą warto zwrócić uwagę jest fakt, że automatyczne różnicowanie wykorzystuje grafy obliczeniowe (jawnie lub nie). Graf obliczeniowy jest reprezentacją wyrażenia matematycznego lub procesu obliczeniowego. Powszechnie używana się go do wizualizacji i zrozumienia przepływu obliczeń związanych z oceną funkcji lub wykonaniem serii operacji.

Większość wzorów matematycznych można podzielić na serię podstawowych operacji arytmetycznych (np. dodawanie, mnożenie, potęgowanie). Aby utworzyć graf obliczeniowy, najpierw tworzymy wierzchołki. Wierzchołki grafu obliczeniowego reprezentują operacje lub funkcje matematyczne. Każdy wierzchołek posiada przypisane do siebie działanie, takie jak dodawanie, mnożenie lub bardziej złożona operacja. Krawędzie w grafie przedstawiają przepływ danych lub zależności pomiędzy operacjami. Krawędź między wierzchołkami wskazuje, że wynik pierwszej operacji jest używany jako wejście dla drugiej operacji. Dane wejściowe na grafie obliczeniowym są zwykle przedstawiane jako wierzchołki bez krawędzi przychodzących, podczas gdy wyjścia to wierzchołki bez krawędzi wychodzących.

Przykład 9. Niech $f(x_1, x_2) = (\cos(\frac{x_1}{x_2}) + \frac{x_1}{x_2} - \sin(x_2))(\frac{x_1}{x_2} - \sin(x_2))$. Graf obliczeniowy takiej funkcji wyglądałby następująco:

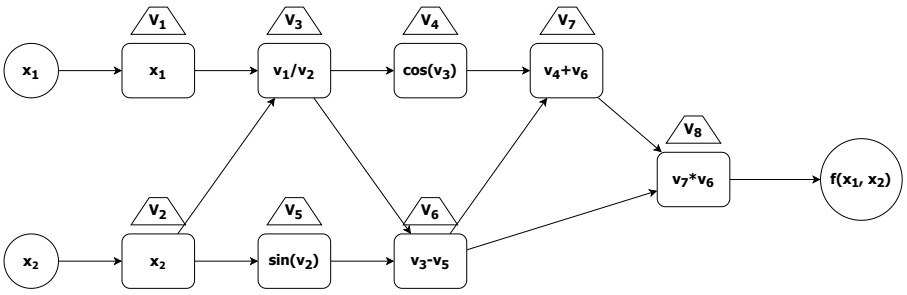


Figure 3.8: Przykładowy graf obliczeniowy funkcji $f(x_1, x_2)$

Kolejną istotną częścią algorytmu jest użycie *liczb podwójnych* (ang. *dual numbers*). Dzięki nim w każdym wierzchołku grafu obliczeniowego nie tylko obliczymy wartości liczbowe funkcji, ale jednocześnie obliczymy ich pochodne. Ten prosty trik pomoże nam ponownie wykorzystać fragmenty obliczone w poprzednich krokach w przyszłych operacjach. Jednocześnie ograniczamy się do obliczania pochodnych tylko prostych operacji arytmetycznych.

W języku programowania **R** możemy zaimplementować takie *liczby podwójne* za pomocą programowania obiektowego:

```
DualNumber <- function(val, eps=0) {
    obj <- list(val = val, eps = eps)
    class(obj) <- "DualNumber"
    return(obj)
}
```

Listing 9: Implementacja liczb podwójnych

Na koniec pozostaje jeszcze kwestia obliczenia wyników poszczególnych operacji na grafie obliczeniowym. Eleganckim sposobem podejścia do tego problemu jest użycie **przeciążanie operatorów** (ang. *operator overloading*). Każdy operator (taki jak „+" lub „-") możemy go wykorzystać do zdefiniowania innego zachowania, specjalnie dostosowanego do naszych *liczb podwójnych*.

Stwórzmy przeciążenia dla każdej z podstawowych operacji. Na początek możemy zacząć od operatora dodawania „+":

```
"+" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val + y$val
    eps <- x$eps + y$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("+")(x, y)
  }
}
```

Listing 10: Przeciążenie operatora dodawania

Każdy operator jest funkcją, która przyjmuje dwa parametry jako dane wejściowe i podaje wynik. Najpierw powinno się rozróżnić wpływ operatora na *liczby podwójne* i inne wartości (dla których działanie operatora pozostanie niezmienione). Jeśli chcemy dodać dwie *liczby podwójne*, musimy dodać zarówno ich wartości, jak i wartości ich pochodnych. Następnie możemy zwrócić nowo utworzony wynik jako *dual number*.

Postępując w podobny sposób możemy zaimplementować przeciążanie operatora dla operatora odejmowania „-":

```
"-" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val - y$val
    eps <- x$eps - y$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("-")(x, y)
  }
}
```

Listing 11: Przeciążenie operatora odejmowania

Przy przeciążaniu operatora mnożenia „*" warto przypomnieć sobie *wzór Leibniza* (wzór służący do wyznaczania pochodnych iloczynu dwóch funkcji):

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x) \tag{3.31}$$

```
"*" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val*y$val
    eps<-y$val*x$eps + x$val*y$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("*")(x, y)
```

```
  }
}
```

Listing 12: Przeciążenie operatora mnożenia

Jako ostatni operator, który opracujemy na potrzeby tej przykładowej implementacji, możemy przeciążyć operatora potęgowania. Pomocny może okazać się wzór na pochodną funkcji potęgowej:

$$f'(x) = \frac{d}{dx}(x^k) = kx^{k-1} \tag{3.32}$$

Należy tylko pamiętać o pomnożeniu wartości uzyskanej z tego wzoru przez obliczoną wcześniej wartość pochodnej:

```
"^" <- function (x, k) {
  if (class(x) == "DualNumber") {
    val <- x$val^k
    eps <- k*x$val^(k-1)*x$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("^")(x, k)
  }
}
```

Listing 13: Przeciążenie operatora potęgowania

Do działania algorytmu automatycznego różniczkowania wystarczą *dual numbers* i przeciążone operatory. Teraz możemy przejść do testowania algorytmu.

Przykład 10. Niech $A = 5$ z uwzględnieniem pierwszej zmiennej, $B = 4$ z uwzględnieniem drugiej zmiennej oraz $C = 7$ z uwzględnieniem trzeciej zmiennej. Ponadto, niech $f(x_1, x_2, x_3) = x_1^2 + x_1x_2 + x_2 + x_3$:

```
# Deklaracja wartosci DualNumer
A <- DualNumber(5, c(1,0,0))
B <- DualNumber(4, c(0,1,0))
C <- DualNumber(7, c(0,0,1))

# Obliczanie wartosci funkcji f(A, B, C).
A^2 + A*B + A + C
```

Listing 14: Przykład automatycznego różniczkowania

Gdy wykonamy te obliczenia, otrzymamy następujące wyniki:

$val

[1] 57

$eps

[1] 15 5 1

attr(,"class")

[1] "DualNumber"

Wykonując prostą operację na liczbach typu *dual number*, możemy otrzymać zarówno wartość funkcji f, jak i wartość pochodnej po każdej ze zmiennych wejściowych. Co więcej, otrzymane wartości dla *$eps* są dokładnie gradientem funkcji f w danym punkcie.

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 4: Metody lokalne

Daniel Kaszyński

4.10 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 11: Pochodna funkcji jednowymiarowej

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

$$f'(x) = \frac{df}{dx}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{4.33}$$

Kiedy mamy do czynienia z funkcją jednowymiarową, domyślnie zakładamy, że wektor h jest po prostu równy 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 12: 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:

$$\frac{df}{dh}(x, h) = \lim_{\delta \to 0} \frac{f(x+\delta h) - f(x)}{\delta} \tag{4.34}$$

$$\frac{df}{dh}(x, h) = \nabla_f(x)h = |\nabla_f(x)||h| \cos(\alpha) \tag{4.35}$$

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. Zasadniczo jest to pochodna funkcji względem jednej z jej zmiennych niezależnych, traktująca pozostałe zmienne jako stałe. Pochodna cząstkowa jest także wrażliwością funkcji.

Definicja 13: 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:

$$\frac{\partial f}{\partial x_i}(x) = \frac{df}{de_i}(x) = \lim_{\delta \to 0} \frac{f(x + \delta e_i) - f(x)}{\delta}$$

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 $h = e_i$.

4.11 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.

$$\mathrm{N}(A) = \mathrm{Null}(A) = \ker(A) = \{\mathbf{x} \in K^n \mid A\mathbf{x} = \mathbf{0}\}. \tag{4.36}$$

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.

$$\ker(A_{n\times n}) = \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \ldots & e_3 \end{array} \\ \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \end{array}$$

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).

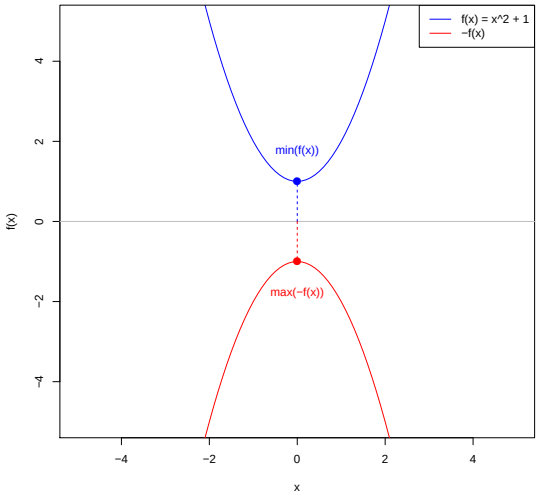
4.12 Optymalność w przestrzeni wielowymiarowej

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 minimalizację funkcji maksymalizacją odwrotności tej funkcji.



$$\underset{x}{\operatorname{argmin}} f(x) = \underset{x}{\operatorname{argmax}}(-f(x)) \tag{4.37}$$

Przykład 11. Niech $f(x, y) = x^2 + 2y^2$ oraz $x_0 = (2, 3)$. Wykres takiej funkcji wyglądałby następująco:

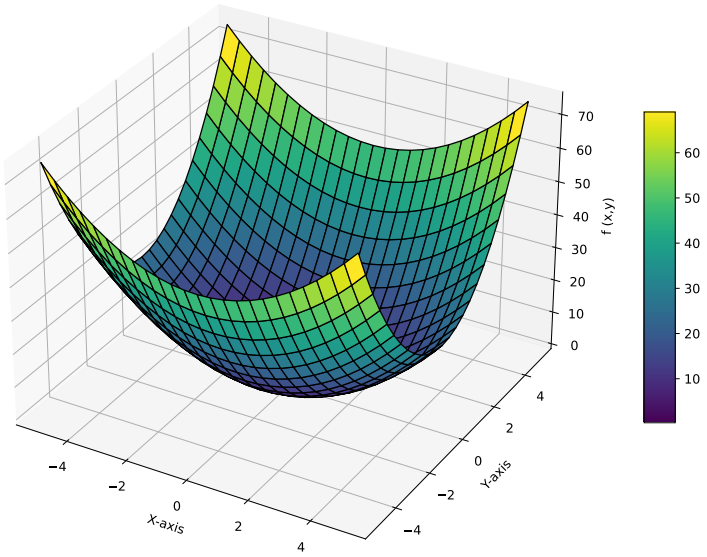


Figure 4.9: Funkcja $f(x) = x^2 + 2y^2$

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

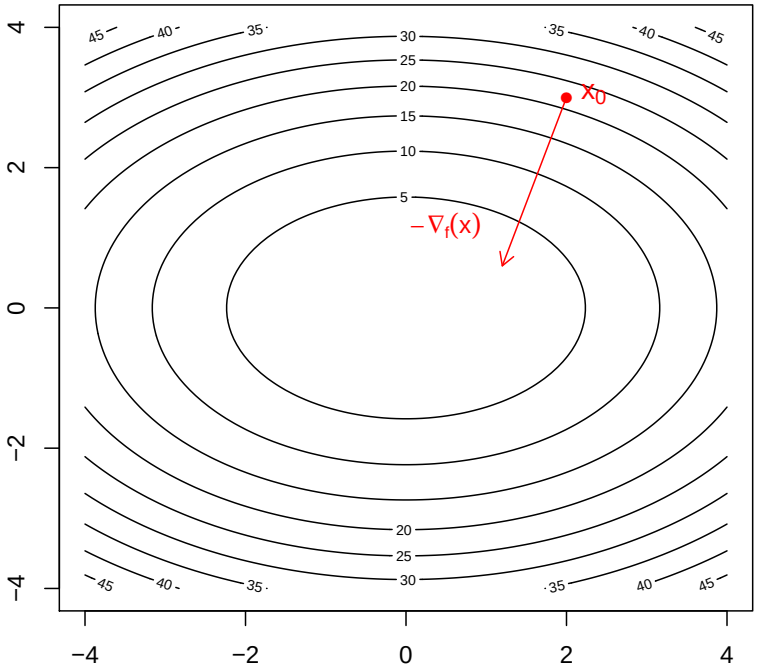


Figure 4.10: Wektor o punkcie zaczepienia x_0, który reprezentuje najszybszy spadek funkcji f

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

$$\operatorname*{argmin}_h \frac{df}{dh}(x,h) = \operatorname*{argmin}_h |\nabla_f(x)||h| \cos(\sphericalangle(\nabla_f(x),h)) \tag{4.38}$$

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 $(\cos(\sphericalangle(\nabla_f(x),h)))$. 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.

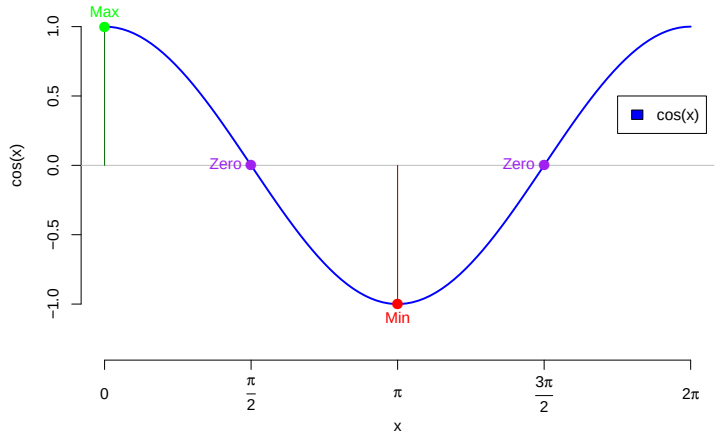


Figure 4.11: Wykres funkcji cosinus

4.13 Metoda spadku gradientu prostego (gradient descent)

Metoda spadku 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 14: Metoda spadku 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:

$$x_{k+1} = x_k - \alpha \nabla_f(x_k) \tag{4.39}$$

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 numerycznie 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 12. 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:

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

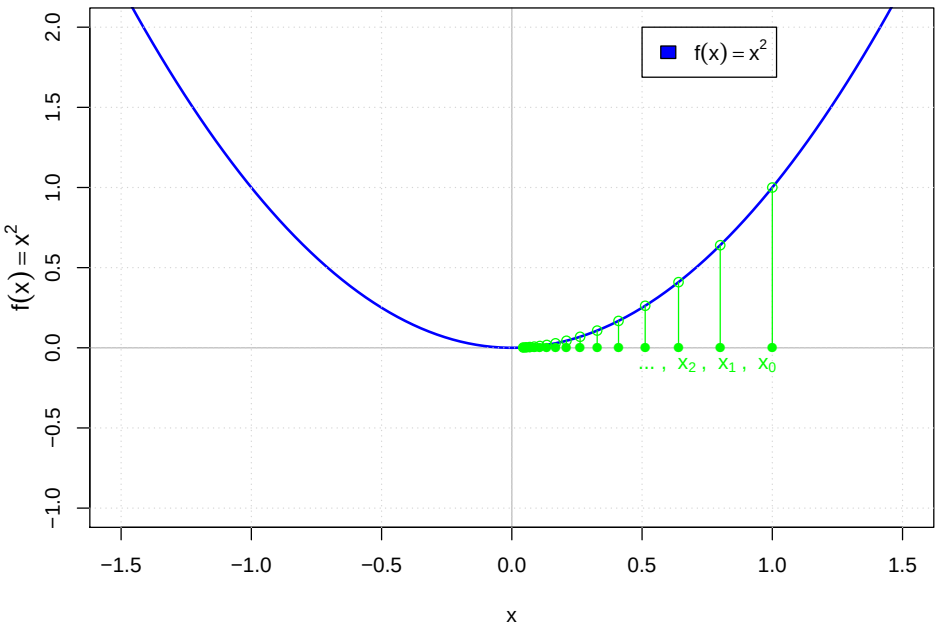


Figure 4.12: Zbieganie kolejnych wartości punktów x_0, x_1, x_2... do ekstremum funkcji $f(x) = x^2$ po ustawieniu parametru $\alpha = 0.1$

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 15: 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 16: 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.

Figure 4.13: 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.

```r
# 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 17: Testowanie gradientu numerycznego

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

```r
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
```

```
37      # jest najlepsze do tej pory
38      if(f(x_new) < results$f_opt){
39        results$x_opt <- x_new
40        results$f_opt <- f(x_new)
41      }
42
43      results$x_hist[k,] <- x_new
44      results$f_hist[k] <- f(x_new)
45
46      x <- x_new
47    }
48    # roznica czasu pomiedzy koncem i poczatkiem algorytmu
49    results$t_eval <- Sys.time() - start_time
50    return(results)
51  }
```

Listing 18: Implementacja metody gradientu prostego

Przykład 13. Niech $f(x_1, x_2) = \frac{1}{8}x_1^2 + x_2^2$ oraz $x_0 = (3, 4)$.

Funkcja f w punkcie x_0 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:

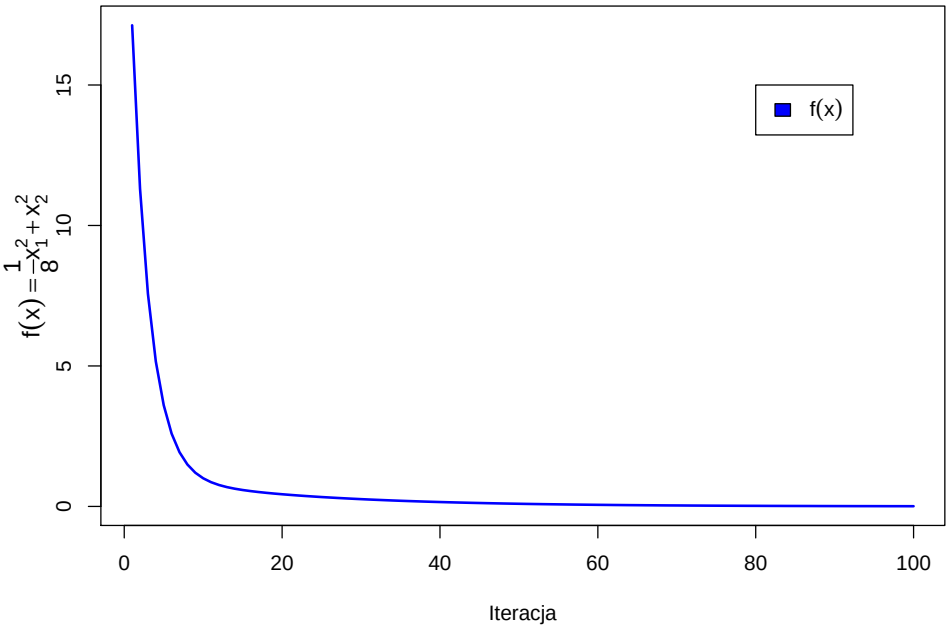


Figure 4.14: Wartości funkcji f stopniowo uzyskiwane podczas 100 iteracji metody gradientu prostego

Możemy wykreślić nie tylko wartości obliczone przez ten algorytm, ale także pozycje uzyskane w kolejnych iteracjach w przestrzeni 2D. Ścieżkę pokonaną przez algorytm metody spadku gradientu

prostego można zobaczyć na poniższym wykresie:

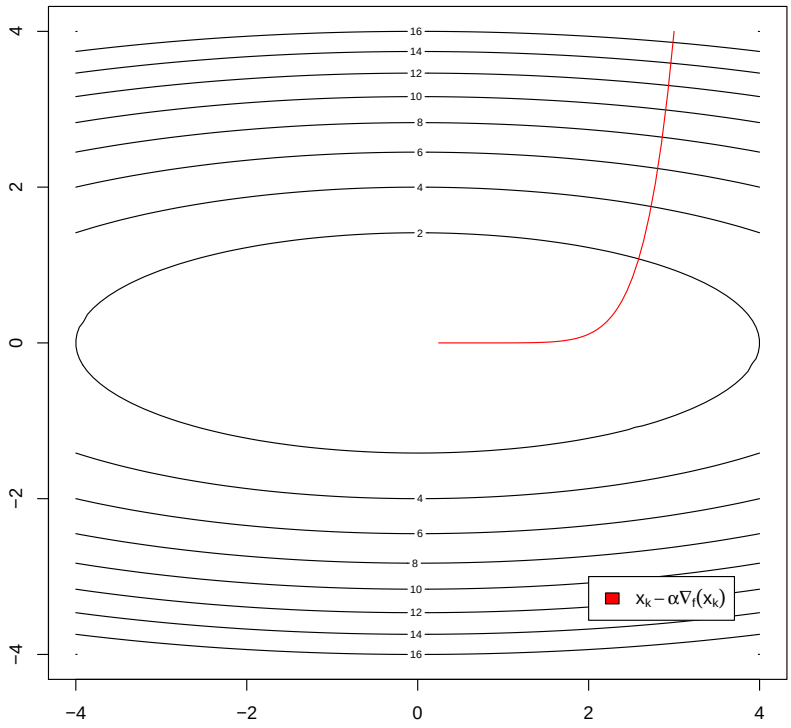


Figure 4.15: Ścieżka przebyta przez algorytm metody gradientu prostego funkcji f w ciągu 100 iteracji w przestrzeni 2D

4.14 Tempo uczenia

Tempo uczenia (ang. *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 14. 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 tempo uczenia α jest równe 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.

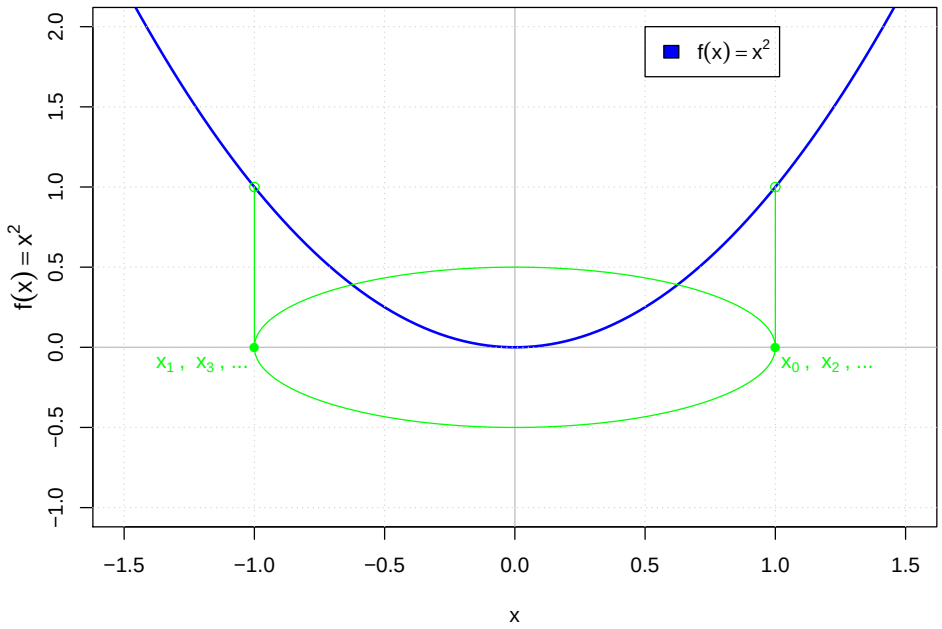


Figure 4.16: Impas powstały w wyniku wykorzystania całej wartości gradientu przy obliczaniu punktów $x_0, x_1, x_2...$

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

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 4 = -3$
- $x_2 = x_1 - \nabla_f(x_1) = -3 - (-108) = 105$
- $x_3 = x_2 - \nabla_f(x_2) = 105 - 4\ 630\ 500 = -4\ 630\ 395$
- $x_4 = x_3 - \nabla_f(x_3) \approx -4\ 630\ 395 - (-3.9711301e + 20) \approx -3.9711301e + 20$ (...)

Jednym z możliwych rozwiązań problemu przedstawionego w Przykładzie 4 jest wprowadzenie pewnego rodzaju parametru 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 15. Niech $f(x_1, x_2) = \frac{1}{8}x_1^2 + x_2^2$ oraz $x_0 = (3, 4)$.

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:

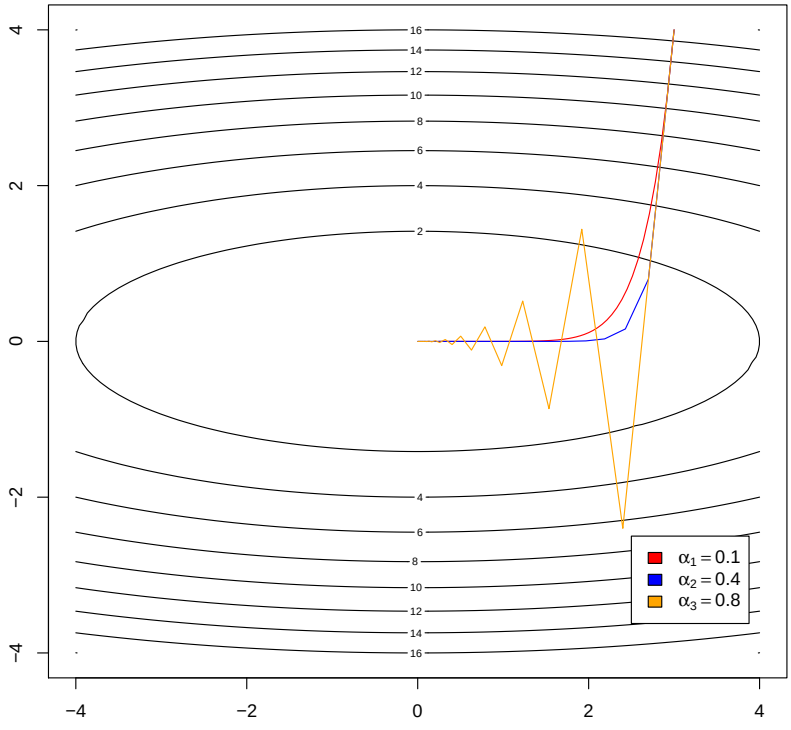


Figure 4.17: Wpływ learning rate na ścieżkę przebytą przez metodę gradientu prostego na funkcji f.

4.15 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).

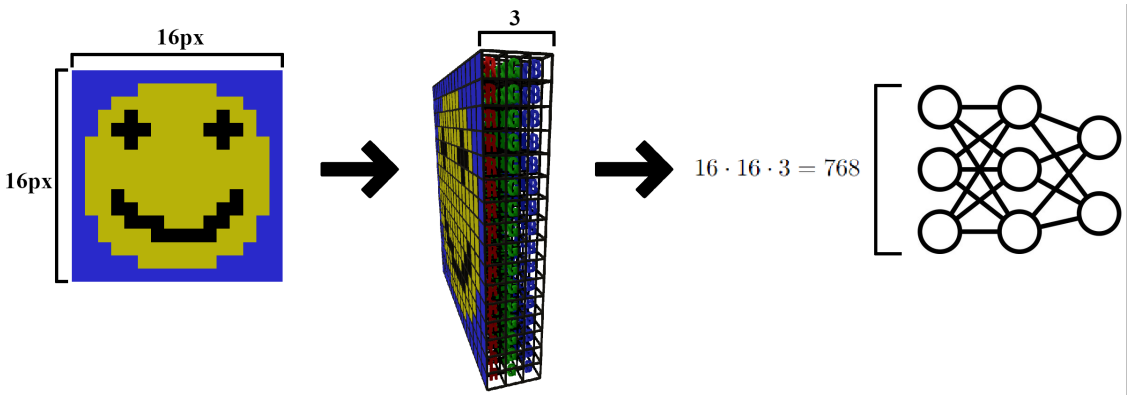


Figure 4.18: Liczba pikseli pomnożona przez 3 kanały RGB jako dane wejściowe dla sieci neuronowej

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:

$$O_j = \sigma\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right) \tag{4.40}$$

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:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.41}$$

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.

$$\nabla C(w_0, w_1, ..., w_n) \tag{4.42}$$

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

4.16 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 tempa uczenia (maksymalną wielkość kroku w kierunku antygradientu), a następnie algorytm sam określa optymalną wartość tego parametru.

Definicja 15: 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:

$$x_{k+1} = x_k - \alpha_{k-best}\nabla_f(x_k) \tag{4.43}$$

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.

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} \frac{d}{d\alpha} \left(x_k - \alpha \nabla_f(x_k)\right) \tag{4.44}$$

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.

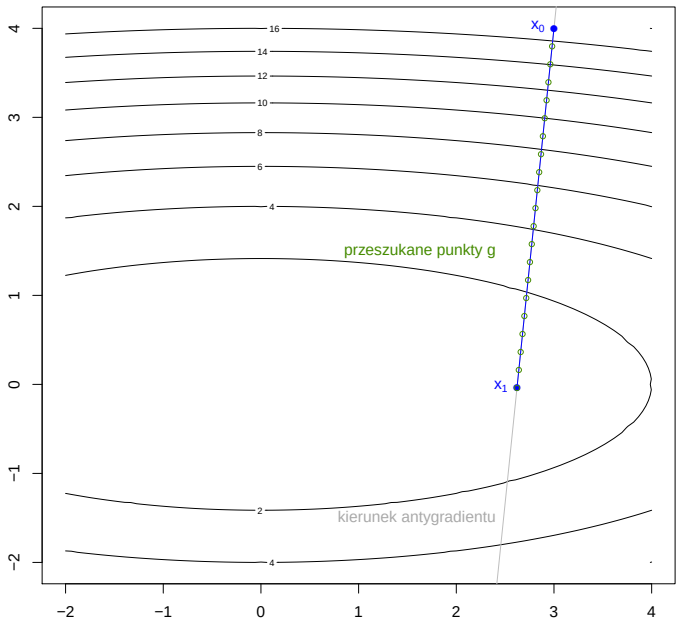


Figure 4.19: Punkty badane podczas jednej iteracji metody najszybszego spadku na funkcji f

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 16. 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):

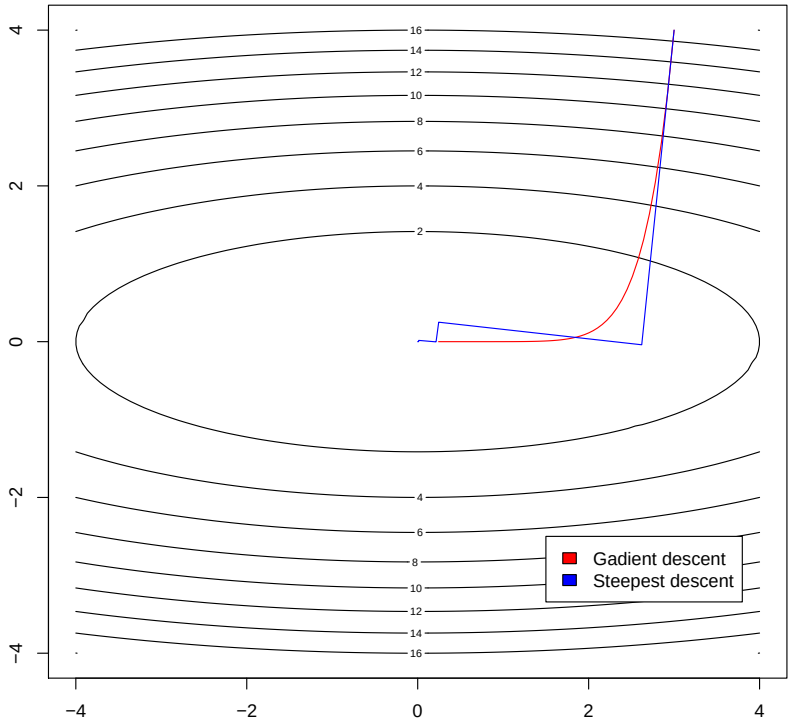


Figure 4.20: Porównanie ścieżki uzyskanej przez metodę najszybszego spadku i metodę gradientu prostego na funkcji f w przestrzeni 2D

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.

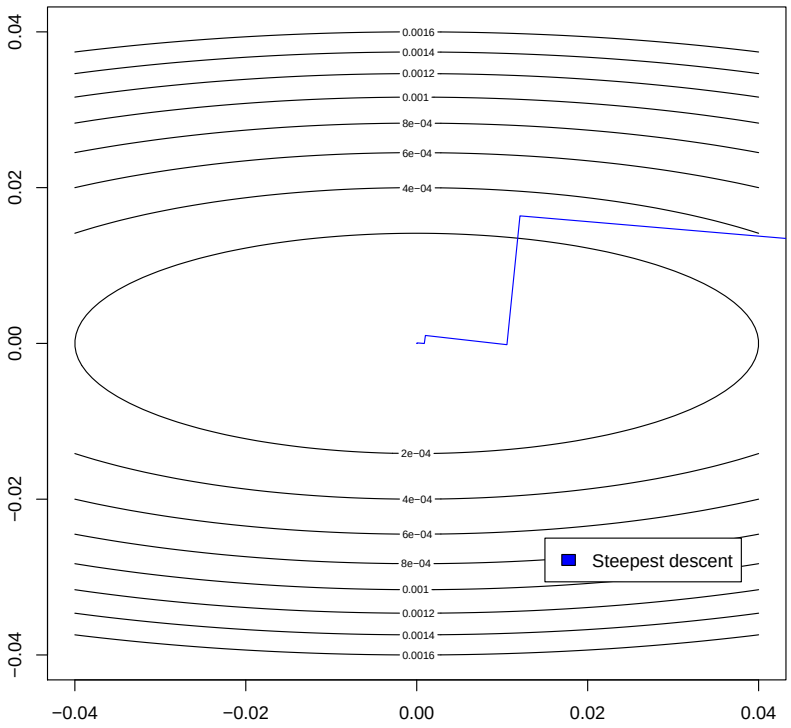


Figure 4.21: Zbliżenie ścieżki uzyskanej przez metodę najszybszego spadku na funkcji f w przestrzeni 2D

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 19: 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)
    }

    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 20: Implementacja metody najszybszego spadku

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 5: Metody wyższego rzędu

Daniel Kaszyński

5.17 Metoda Newtona

5.17.1 Teoria stojąca za metodą Newtona

Metoda Newtona (ang. *Newton Descent*) to kolejny algorytm iteracyjny, często używany do minimalizacji funkcji, podobnie jak metoda gradientu prostego i metoda najszybszego spadku omawiane w poprzednich wykładach. Wykorzystuje aproksymacje drugiego rzędu do znalezienia punktów krytycznych danej funkcji f. Podstawową ideą metody Newtona w optymalizacji jest iteracyjne aktualizowanie wstępnego przypuszczenia dotyczącego optymalnego rozwiązania w oparciu o pierwszą i drugą pochodną funkcji. Reguła aktualizacji wywodzi się z rozwinięcia funkcji w szereg Taylora wokół bieżącego przypuszczenia.

Aby lepiej zrozumieć działanie tego algorytmu, zacznijmy od przybliżenia szeregu Taylora wokół punktu x_k aż do pochodnej drugiego stopnia (przybliżenie Taylora drugiego rzędu f):

$$f(x_k + h) \approx f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2 \tag{5.45}$$

Celem tej metody jest znalezienie h, dla którego funkcja wokół danego punktu x_k zmienia się najszybciej. Zakładamy, że w każdym kroku podane jest x_k, a zatem jest to wartość stała. Mając to na uwadze możemy dokonać obserwacji, iż szereg Taylora jest wielomianowym wzorem na aproksymację wartości ($f(x_k)$, $f'(x_k)$ oraz $\frac{1}{2}f''(x_k)$ są stałe, pozostawiając nam do dyspozycji tylko h).

Następnie możemy zastosować warunki pierwszego rzędu i przyrównać pochodną naszego przybliżenia do zera:

$$\frac{d}{dh}\left(f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2\right) = f'(x_k) + f''(x_k)h = 0 \tag{5.46}$$

Następnie, stosując proste przekształcenie wzoru, możemy otrzymać wzór na optymalną wartość h:

$$h = -\frac{f'(x_k)}{f''(x_k)} \tag{5.47}$$

Ten wzór na h można wykorzystać do obliczenia kroków podejmowanych w kolejnych iteracjach metody Newtona dla funkcji 1D.

Definicja 16: Metoda Newtona (ang. *Newton Descent*) dla funkcji 1D

Mając na uwadze fakt, iż metoda Newtona jest algorytmem iteracyjnym, punktem obliczonym w $(k-1)$-tej iteracji tego algorytmu dla funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ nazywamy:

$$x_{k+1} = x_k + h = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{5.48}$$

gdzie $k \in \mathbb{N}$ jest numerem iteracji, $f'(x_k)$ jest pochodną pierwszego stopnia funkcji f w punkcie x_k oraz $f''(x_k)$ jest pochodną drugiego stopnia funkcji f w punkcie x_k.

Algorytm Newton descent przedstawiony we wzorze powyżej (5.48) można uogólnić do większej ilości wymiarów, zastępując pochodną gradientem, a odwrotność drugiej pochodnej odwrotnością Hesjanu (ponieważ jak wiemy z poprzednich wykładów, Hesjan jest uogólnieniem drugiej pochodnej).

Definicja 17: Metoda Newtona (ang. *Newton Descent*)

Mając na uwadze fakt, iż metoda Newtona jest algorytmem iteracyjnym, punktem obliczonym w $(k-1)$-tej iteracji tego algorytmu dla funkcji $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ nazywamy:

$$x_{k+1} = x_k - H_f(x_k)^{-1} \nabla_f(x_k) \tag{5.49}$$

gdzie $k \in \mathbb{N}$ jest numerem iteracji, $H_f(x_k)^{-1}$ jest odwrotnością Hesjanu a $\nabla_f(x_k)$ jest gradientem funkcji f obliczoną w punkcie x_k.

Uwaga! Używając metody Newtona, domyślnie nie wiemy, czy optymalizujemy funkcje w kierunku minimum, maksimum czy innych punktów krytycznych danej funkcji. Newton Descent kieruje się w stronę punktu stacjonarności funkcji. Dlatego dobrą praktyką jest wykorzystanie metody Newtona jako ostatniego kroku optymalizacji, aby szybciej dojść do ekstremum funkcji, do której doszliśmy wcześniej innym algorytmem.

Przykład 17. Niech $f(x) = ax^2 + bx + c$ oraz niech x_0 będzie punktem startowym metody Newtona.

Korzystając ze wzoru na metodę Newtona dla podanej funkcji 1D f, otrzymujemy:

$$x_1 = x_0 - \frac{2ax_0 + b}{2a} \tag{5.50}$$

Możemy wówczas podzielić ułamek uzyskany po prawej stronie równania na dwie części:

$$x_1 = x_0 - \frac{2ax_0}{2a} - \frac{b}{2a} \tag{5.51}$$

Dzięki tej transformacji możemy wyeliminować wyrażenia zawierające x_0:

$$x_1 = -\frac{b}{2a} \tag{5.52}$$

Otrzymany wzór na x_1 jest wzorem na wierzchołek funkcji wielomianowej drugiego rzędu (w tym przypadku minimum). Zatem w jednym kroku jesteśmy w stanie znaleźć ekstremum funkcji. Jak widzimy, algorytm ten jest bardzo skuteczny, gdy mamy do czynienia z problemami związanymi z formami kwadratowymi.

Jak widać we wzorze na Newton descent (5.49), aby obliczyć kolejne kroki wykonywane w każdej iteracji algorytmu, musimy znać Hesjan funkcji. Aby móc obliczyć Hesjan funkcji, należy wyprowadzić pochodną

cząstkową po x_i oraz x_j.

$$H_f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Aby obliczyć Hesjan numerycznie, musimy zastosować przybliżenia pochodnych cząstkowych - różnic skończonych. Oznacza się je za pomocą znaku Δ. Dlatego macierz Hessego wykorzystująca wspomniane przybliżenia wyglądałaby następująco:

$$H_f(x) = \begin{bmatrix} \frac{\Delta^2 f}{\Delta x_1^2} & \cdots & \frac{\Delta^2 f}{\Delta x_1 \Delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\Delta^2 f}{\Delta x_n \Delta x_1} & \cdots & \frac{\Delta^2 f}{\Delta x_n^2} \end{bmatrix}$$

Mając na uwadze wzór na różnicę centralną, możemy zapisać pojedynczą skończoną różnicę po x_i jako:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{\Delta f}{\Delta x_i}(x) = \frac{f(x + e_i h) - f(x - e_i h)}{2h} \tag{5.53}$$

Podobnie, każdy element macierzy Hessego, będący skończoną różnicą pomiędzy x_i i x_j, będzie wyglądał następująco:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j}(x) = \frac{f(x + e_i h + e_j h) - f(x + e_i h - e_j h) - f(x - e_i h + e_j h) + f(x - e_i h - e_j h)}{4h^2} \tag{5.54}$$

Warto zaznaczyć, że kolejność stosowania przybliżeń pochodnych cząstkowych nie ma znaczenia. Jeśli funkcja $f \in C^2$ (jest co najmniej dwukrotnie różniczkowalna), korzystając z twierdzenia Schwartza możemy wskazać, że:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j} = \frac{\Delta^2 f}{\Delta x_j \Delta x_i} \tag{5.55}$$

co również świadczy o:

$$H_f(x) = H_f^T(x) \tag{5.56}$$

Kolejną rzeczą wartą uwagi jest fakt, iż nie z każdej funkcji można łatwo uzyskać Hesjan (na przykład funkcje, które nie mają krzywizny). Aby zapewnić prawidłową pracę algorytmu, należy zastosować pewien trik. Ten trik sprowadza się do korekcji Hesjanu funkcji poprzez dodanie do niego macierzy diagonalnej pomnożonej przez małą wartość λ. Zmianę tę należy zastosować zawsze, gdy wyznacznik $abs(H_f(x))$ jest mniejszy niż odgórnie ustalony próg t. Opisana metoda jest często nazywana **regresją grzbietu** lub **korektą Levenberga-Marquardta**.

5.17.2 Metoda Newtona-Raphsona

Warto również wspomnieć o innym algorytmie, podobnym do metody Newtona. Nazywa się on metodą Newtona-Raphsona (a czasami po prostu metodą Newtona). Algorytm ten jest algorytmem znajdowania miejsc zerowych, który generuje sukcesywnie lepsze przybliżenia tych miejsc dla danej funkcji. Wzór używany przez ten algorytm wygląda następująco:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{5.57}$$

Uwaga! Jest to wzór bardzo podobny do wzoru metody Newtona, różni się jednak stopniem pochodnych funkcji f. Dwa wspomniane algorytmy próbują rozwiązać dwa różne problemy. Łączy ich jednak wspólna relacja. Metoda Newtona-Raphsona znajduje miejsca zerowe podanej jej funkcji. Zakładając, że dana funkcja jest już pochodną innej funkcji (branej pod uwagę przez algorytm Newton descent), znalezienie tych miejsc zerowych w istocie daje nam wzór na Newton descent i rozwiązuje adresowane przez niego problemy.

5.17.3 Implementacja algorytmu Newton descent

Używając języka programowania **R** możemy napisać implementację metody Newtona. Po pierwsze potrzebujemy funkcji do obliczenia numerycznej macierzy Hessego:

```
num_hessian <- function(f, x, h = 10^-3){
  #' Numeryczna Macierz Hessego
  #'
  #' @description Funkcja odpowiedzialna za wyznaczenie numerycznego Hesjanu
  #' poprzez obliczenie macierzy pochodnych czastkowych.
  #'
  #' @param f funkcja. Funkcja, ktorej Hesjan wyznaczamy
  #' @param x wektor numeryczny. Punkt, w ktorym wyznaczany jest
  #' numeryczny Hesjan
  #' @param h skalar. Wartosc roznicy skonczonej
  #'
  #' @usage num_hessian(f, x)
  #'
  #' @return Hesjan pochodnych czastkowych funkcji f.

  n <- length(x)
  H <- matrix(NA, nrow = n, ncol = n)
  E <- diag(n)

  for(i in 1 : n) { # Wiersze
      for(j in 1 : n) { # Kolumny
          H[i, j] <- (
                f(x+E[i,]*h+E[j,]*h) - f(x+E[i,]*h-E[j,]*h)
                - f(x-E[i,]*h+E[j,]*h) + f(x-E[i,]*h-E[j,]*h)
            ) / (4*h^2)
      }
  }

  return(H)
}
```

Listing 21: Implementacja numerycznej macierzy Hessego

Wzór stosowany w kolejnych iteracjach algorytmu Newton descent wymaga obliczenia odwrotności macierzy Hessego. Na szczęście język **R** zapewnia gotowe rozwiązanie tego problemu w postaci metody *solve()*.

```
my_fun <- function(x) 1/8*x[1]^2+x[2]^2
x0 <- c(3, 4)
solve(num_hessian(my_fun, x0)) # Zwraca odwrotnosc macierzy Hessego
```

Listing 22: Przykład metody solve() dla Hesjanu funkcji

Having the ability to calculate the inverse of the matrix, we can finally create the code necessary for the Newton descent algorithm. Example of it's implementation can look like this:

Mając możliwość obliczania odwrotności macierzy, możemy teraz stworzyć kod algorytmu metody Newtona. Przykład implementacji może wyglądać następująco:

```
newton_descent <- function(f, x, K = 100){
```

```
  #' Metoda Newtona
  #'
  #' @description Funkcja odpowiedzialna za algorytm metody Newtona.
  #'
  #' @param f: funkcja celu
  #' @param x: punkt startowy
  #' @param K: maksymalny limit iteracji
  #'
  #' @returns
  #' Lista wynikow zawiera natepujace elementy:
  #' * x_opt: znalezione rozwiazanie
  #' * f_opt: wartosc funkcji celu w znalezionym rozwiazaniu
  #' * x_hist: historia eksplorowanych rozwiazan
  #' * f_hist: historia wartosci funkcji celu
  #' * t_eval: czas dzialania 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){
    # obliczanie gradientu i hesjanu funkcji f
    G <- grad(f, x)
    H <- num_hessian(f, x)

    # uzycie regresji grzbietowej w celu rozwiazania
    # sytuacji, w ktorych nie mozna obliczyc hesjanu
    if(abs(det(H)) < 10^(-3)) H <- H + diag(n)*10^(-3)

    # opis przejscia z punktu x_k do x_k+1
    x_new <- x - solve(H) %*% G

    # sprawdzenie czy nowe rozwiazanie
    # jest najlepszym dotychczas
    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 startem algorytmu
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 23: Przykładowa implementacja Newton descent

Przykład 18. Niech $f(x_1, x_2) = \frac{1}{8}x_1^2 + x_2^2$ oraz $x_0 = (3, 4)$.

Zbieganie metody Newtona można zobaczyć na poniższym wykresie (wraz ze ścieżkami utworzonymi przez inne podobne algorytmy):

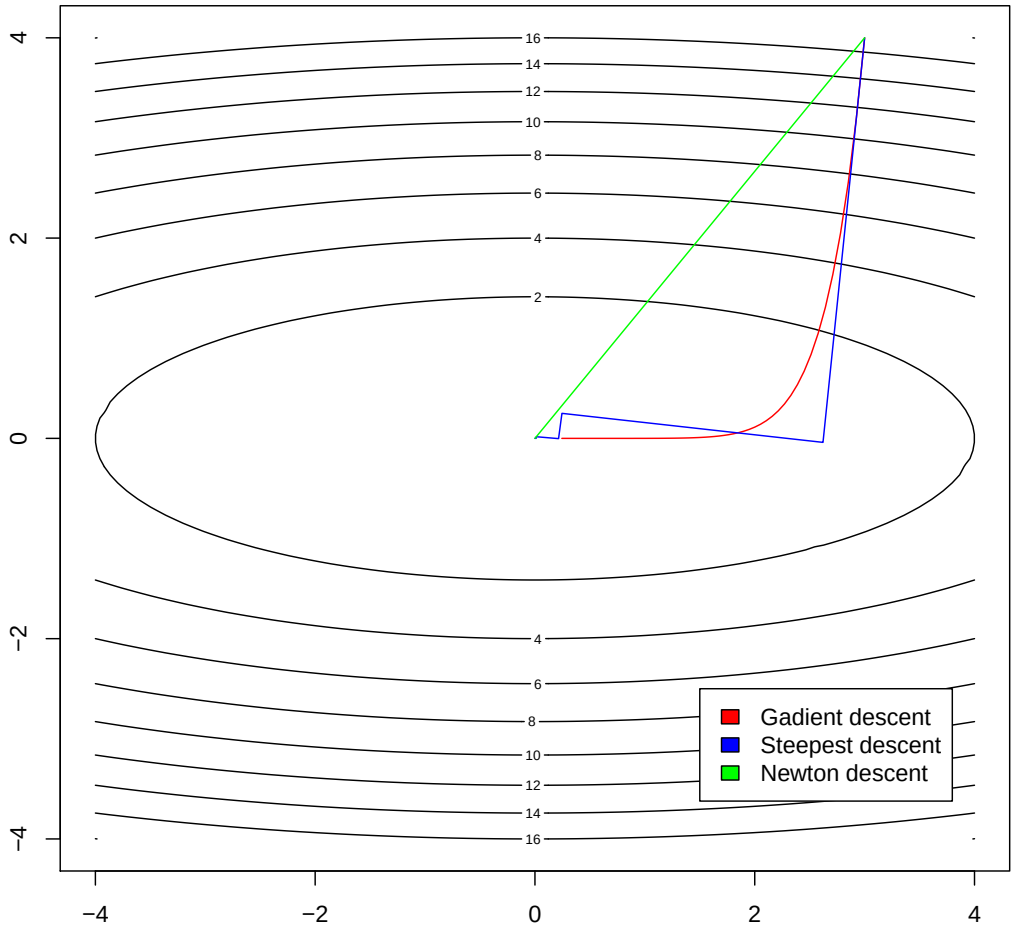


Figure 5.22: Porównanie ścieżek uzyskanych przez metodę Newtona i inne podobne algorytmy na funkcji f w przestrzeni 2D

Możemy zaobserwować, że metoda Newtona ma ścieżkę znacznie różniącą się od pozostałych algorytmów. Metoda najszybszego spadku i metoda gradientu prostego poruszają się ortogonalnie w odniesieniu do wykresu przeciwnego. Newton descent zmierza znacznie bardziej bezpośrednio w kierunku globalnego minimum funkcji. Nie kieruję się już w stronę lokalnie najlepszego rozwiązania. To zachowanie jest spowodowane faktem, że funkcja f jest funkcją kwadratową.

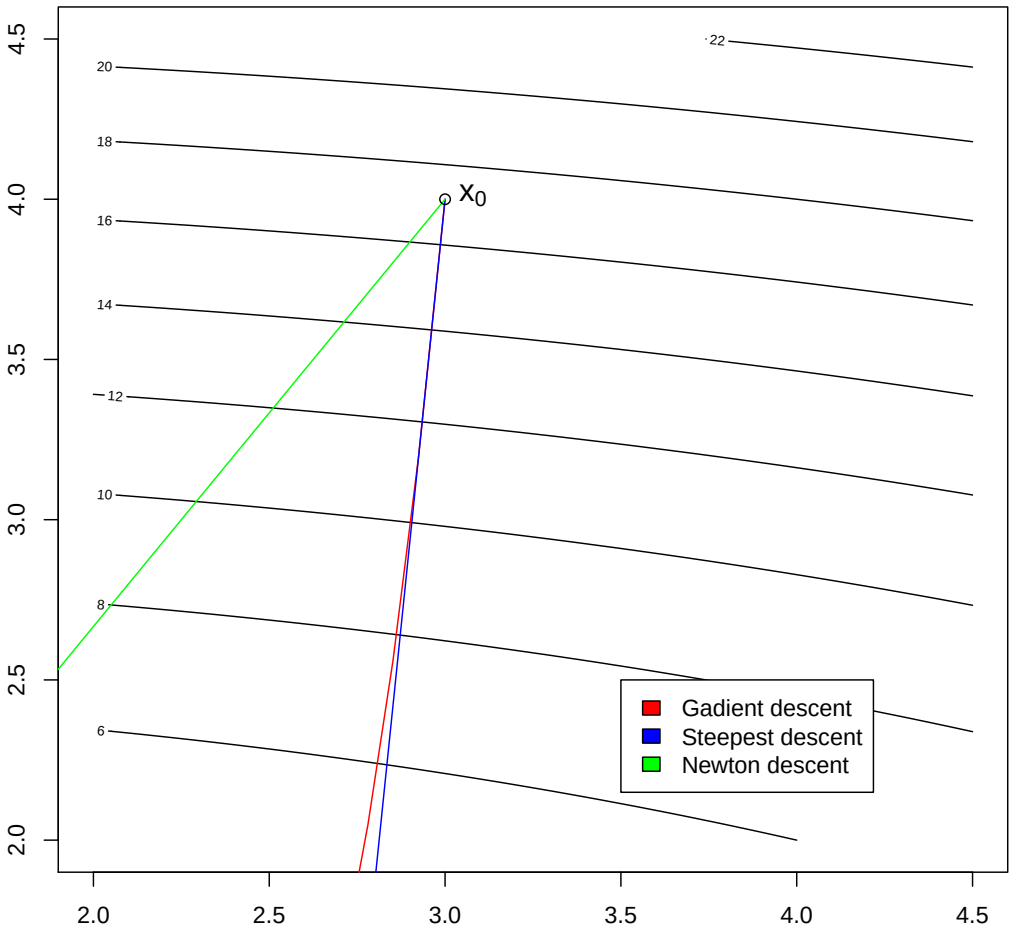


Figure 5.23: Zbliżenie ścieżek uzyskanych metodą Newtona i innych podobnych algorytmów

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 6: Simulated Annealing

Daniel Kaszyński

6.18 Testowanie algorytmów optymalizacyjnych

Przy pracy nad metodami optymalizacyjnymi nieraz zdarza się, iż musimy dokładnie przetestować działanie tworzonych przez nas algorytmów. Proste funkcje tylko do pewnego stopnia zapewniają nam możliwość sprawdzenia działania naszych rozwiązań. Chcąc sprawdzić charakterystyki metod optymalizacji na trudniejszych przypadkach warto jest sięgnąć po bardziej złożone funkcje.

Istnieje wiele złożonych funkcji, na których można testować złożone algorytmy optymalizacyjne. Lista przykładowych funkcji testowych umieszczona jest między innymi na stronie Wikipedii (https://en.wikipedia.org/wiki/Test_functions_for_optimization).

Jedną z popularnych funkcji do testowania algorytmów optymalizacyjnych jest funkcja Schaffer-a. Wyrażana jest ona przy pomocy wzoru:

$$f(x_1, x_2) = 0.5 + \frac{sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 + y^2)]^2} \tag{6.58}$$

Funkcja ta posiada minimum w punkcie $f(0, 0) = 0$ i przyjmuje wartości z zakresu $< 0, 1 >$.

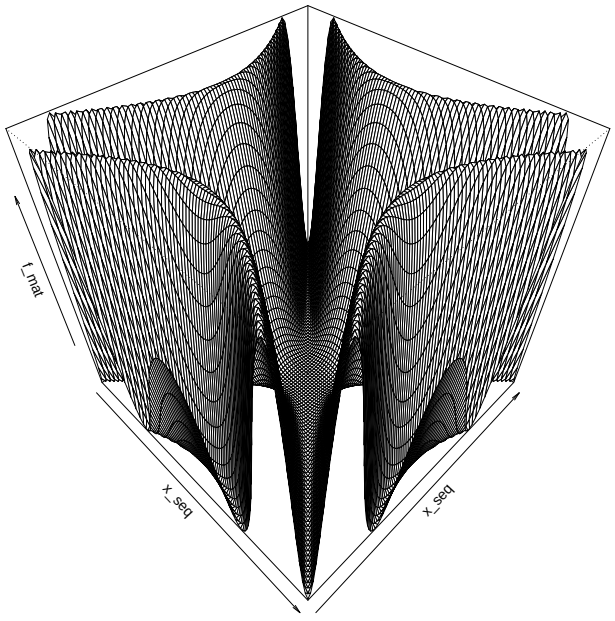


Figure 6.24: Wykres funkcji Schaffer-a w przestrzeni 3D.

Na wykresie funkcji możemy zaobserwować, iż w okolicach punktu $(0, 0)$ znajduje się dolina zawierająca ekstremum globalne funkcji. Dolina ta przybiera kształt "X" i otoczona jest drastycznymi skokami w wartościach funkcji. Przestrzeń leżąca za tymi skokami charakteryzuje się występowaniem spadków i wzrostów wartości funkcji we wzorcu przypominającym falowanie. Warto również zauważyć, iż warstwice fal leżą prawie że równolegle do siebie nawzajem.

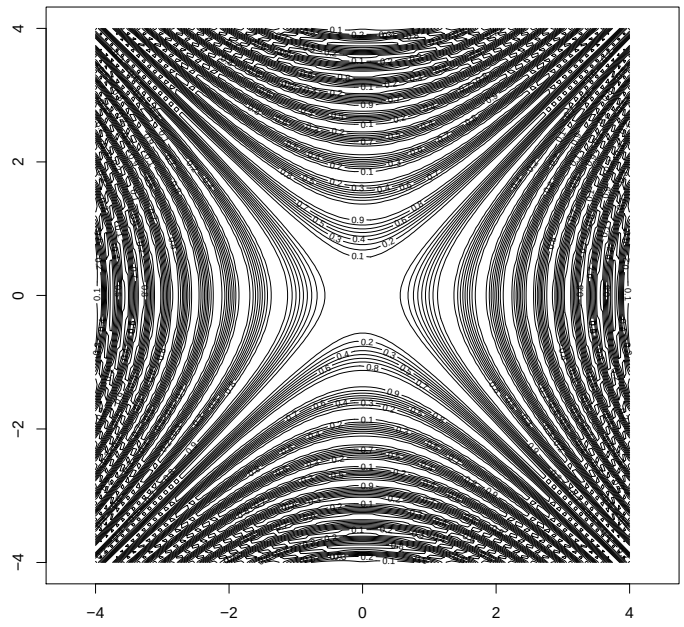


Figure 6.25: Wykres warstwicy funkcji Schaffer-a.

Funkcja ta znakomicie nadaje się do testowania zdolności algorytmów do eksploracji różnych sekcji przestrzeni funkcji w celu odnalezienia globalnego ekstremum. Funkcja Schaffer-a posiada zróżnicowaną topografię. Bez odpowiedniej eksploracji, algorytmy mogą utknąć w ekstremum lokalnym. Może to prowadzić do suboptymalnych rozwiązań, które nie są globalnie najlepsze.

Aby uniknąć utknięcia w ekstremum lokalnym, można zastosować kilka różnych technik eksploracyjnych. Należą do nich między innymi metody stochastyczne, czyli algorytmy oparte na losowości, takie jak algorytmy ewolucyjne czy algorytmy genetyczne. Innymi sposobami mogą być metody wielopunktowe, populacyjne czy adaptacyjne strategie eksploracyjne.

6.19 Testy algorytmów przeszukiwania lokalnego

Na początek warto ustalić, jak na funkcji Schaffer-a zachowują się poznane przez nas na poprzednich wykładach algorytmy optymalizacyjne, takie jak algorytm metody gradientu prostego (gradient descent) czy metoda najszybszego spadku (steepest descent).

Przykład 19. Przyjmijmy jako rozpatrywaną funkcję f funkcję Schaffer-a oraz punkt startowy $x_{start} = (3, 4)$.

Ponadto, przyjmijmy jako paramety metody gradientu prostego wartości learning rate $\alpha = 0.02$ oraz maksymalny limit iteracji $K = 30000$. Podobnie przy metodzie najszybszego spadku załóżmy, że maksymalny learning rate $\alpha = 5$, ilość iteracji przeszukiwania najlepszego learning rate $g = 1000$ oraz maksymalny limit iteracji $K = 30000$.

Przy podjęciu próby optymalizacji funkcji z punktu startowego x_{start} przy pomocy algorytmu metody gradientu prostego oraz metody najszybszego spadku udało się zbliżyć do ekstremum lokalnego funkcji. Ścieżki uzyskane w kolejnych krokach iteracji tych algorytmów możemy zaobserwować na wykresie poniżej:

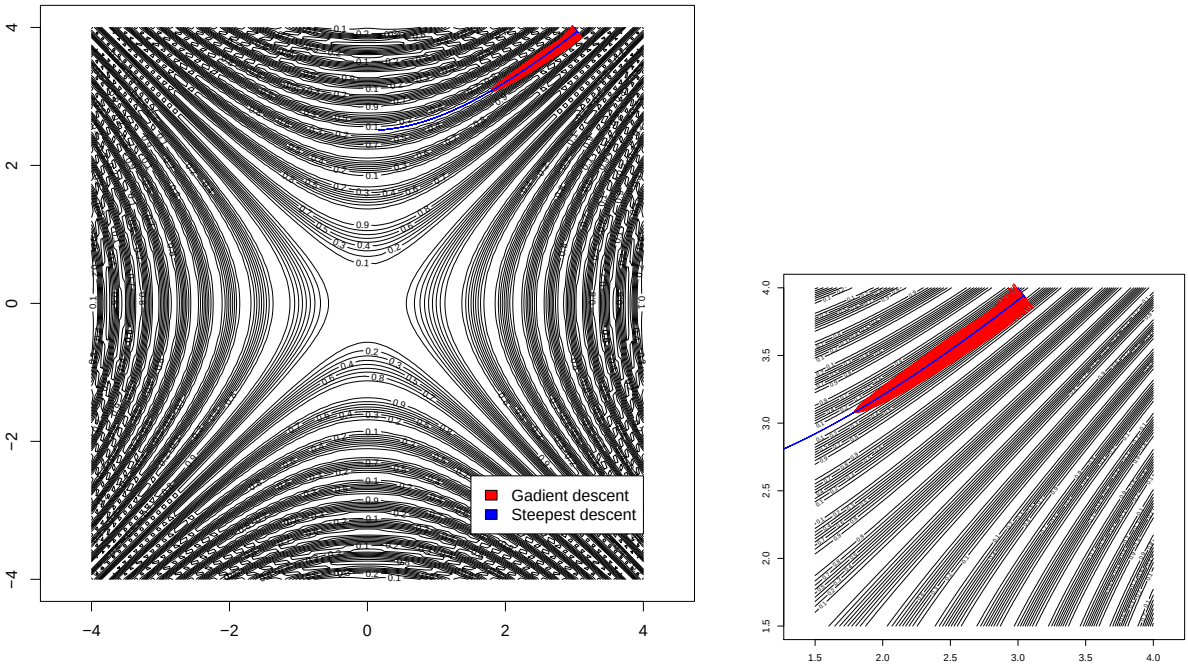


Figure 6.26: Ścieżki stworzone przez algorytmy gradient descent oraz steepest descent na bazie funkcji Schaffer-a.

Łatwo można zaobserwować, iż żadnemu z algorytmów nie udało się wydostać z początkowej doliny, w której okolicy znajdował się punkt startowy x_{start}. Poruszały się one bardzo wolno i wymagały przy tym dużej liczby iteracji. Co więcej, utknęły one dążąc do lokalnego minimum funkcji. Takie zjawisko nazywa się pułapką lokalnego ekstremum. Na podstawie tego można wnioskować, że gradient descent oraz steepest descent nie posiadają własności przeszukiwania globalnego (ang. *global search*). W przypadku takich funkcji jak funkcja Schaffer-a warto jest wprowadzić mechanizmy ucieczki pozwalające na przeszukiwanie innych regionów.

6.20 Simulated Annealing

Symulowane wyżarzanie (ang. *simulated annealing*) to probabilistyczny algorytm optymalizacji inspirowany procesem wyżarzania w metalurgii. W metalurgii wyżarzanie jest techniką stosowaną w celu zmniejszenia de-

fektów i poprawy struktury krystalicznej materiałów poprzez ich ogrzewanie, a następnie stopniowe chłodzenie. Podobnie w kontekście optymalizacji symulowane wyżarzanie służy do znalezienia globalnego minimum funkcji poprzez naśladowanie procesu stopniowego chłodzenia.

Pierwszym krokiem algorytmu *simulated annealing* jest jego inicjalizacja. W tej fazie ustalane są parametry startowe algorytmu, do których należą:

- f - rozpatrywana funkcja celu,
- x_0 - punkt startowy algorytmu,
- d - otoczenie,
- t_0 - temperatura początkowa,
- α - spadek temperatury,
- K - liczba iteracji algorytmu,

oraz parametry wewnętrzne takie jak:

- A_k - wartość funkcji aktywacji,
- t_k - temperatura w każdej iteracji.

Symulowane wyżarzanie wykorzystuje parametr temperatury, który kontroluje prawdopodobieństwo przyjęcia gorszych rozwiązań w miarę postępu algorytmu. Początkowo temperaturę ustawia się na wysoką wartość i stopniowo zmniejsza się ją w kolejnych iteracjach, zgodnie z przyjętą strategią wyżarzania.

W każdej iteracji algorytmu generowane jest rozwiązanie sąsiednie. Tego sąsiada uzyskuje się dokonując niewielkiej losowej zmiany w bieżącym rozwiązaniu w obrębie otoczenia d. Potem następuje ocena funkcji celu dla bieżącego rozwiązania oraz sąsiedniego rozwiązania. Jeżeli sąsiednie rozwiązanie jest lepsze (tj. ma niższą wartość funkcji celu), akceptuje się je jako nowe, aktualne rozwiązanie. Jeżeli rozwiązanie sąsiada jest gorsze, należy je zaakceptować z prawdopodobieństwem określonym przez funkcję aktywacji A_k i aktualną temperaturę t_k. Ta funkcja aktywacji (lub inaczej prawdopodobieństwa) pozwala algorytmowi czasami akceptować gorsze rozwiązania na początku procesu optymalizacji, co pomaga zapobiegać utknięciu algorytmu w lokalnych minimach. Następnie obniżana jest temperatura zgodnie z przyjętą strategią. Wraz ze spadkiem temperatury maleje prawdopodobieństwo przyjęcia gorszych rozwiązań i zwiększa się prawdopodobieństwo zbieżności algorytmu w stronę minimum globalnego. Kroki te powtarzane są w kolejnych iteracjach, aż do spełnienia kryterium zatrzymania, jekim może być osiągnięcie maksymalnej liczby iteracji lub osiągnięcie zadowalającego rozwiązania.

Symulowane wyżarzanie jest skuteczne w znajdowaniu niemal optymalnych rozwiązań złożonych problemów optymalizacyjnych, w których tradycyjne metody oparte na gradiencie mogą sprawiać problemy, szczególnie gdy funkcja celu jest niewypukła lub zaszumiona. Pozwalając algorytmowi okazjonalnie akceptować gorsze rozwiązania, symulowane wyżarzanie może zbadać szerszy zakres przestrzeni rozwiązań i uniknąć uwięzienia w lokalnych minimach.

Z wykorzystaniem języka programowania **R** możemy napisać własną implementację algorytmu *simulated annealing*. Przykładowa implementacja tej metody może wyglądać następująco:

```
simulated_annealing <- function(f, x0, d, t0, a, K = 100){
  #' Symulowane Wyzarzanie
  #'
  #' @description Funkcja odpowiedzialna za aproksymacje ekstremum globalnego
  #' dla funkcji f w K krokachprzy pomocy algorytmu symulowanego wyzarzania
  #' (ang. simulated annealing).
```

```
  #'
  #' @param f funkcja. Funkcja celu algorytmu.
  #' @param x0 wektor numeryczny. Punkt startowy algorytmu.
  #' @param d skalar. Badane otoczenie.
  #' @param t0 skalar. Temperatura poczatkowa.
  #' @param a skalar. Parametr okreslajacy tempo spadku temperatury.
  #' @param K skalar. Opcjonalny parametr okreslajacy maksymalny limit iteracji (domyslnie
    100).
  #'
  #' @usage simulated_annealing(f, x0, d, t0, 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()
  n <- length(x0)
  results <- list(x_opt = x0,
                  f_opt = f(x0),
                  x_hist = matrix(NA, nrow = K, ncol = n),
                  f_hist = rep(NA, K),
                  A_k = rep(NA, K),
                  t_k = rep(NA, K),
                  t_eval = NA)

  results$x_hist[1,] <- x0
  results$f_hist[1] <- f(x0)

  x <- x0
  t_k <- t0
  for(k in 2:  K){
    # opis wyboru sasiedniego rozwiazania z otoczenia
    x_c <- x + runif(n, min = -d, max = d)
    A_k <- min(1, exp(-(f(x_c) - f(x)) / (t_k)))

    # sprawdzenie, czy nowe rozwiazanie
    # powinno zostac przyjete jako nowe
    # oraz czy jest najlepsze do tej pory
    if(runif(1) < A_k){
      x <- x_c
      if(f(x) < results$f_opt){
        results$x_opt <- x
        results$f_opt <- f(x)
      }
    }

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

    results$A_k[k] <- A_k
    results$t_k[k] <- t_k

    t_k <- t_k*a
  }

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

Listing 24: Implementacja algorytmu *simulated annealing*

Warto zwrócić uwagę na to, że do wyboru sąsiedniego rozwiązania z otoczenia wykorzystano funkcję *runif()*. Funkcja ta dostarcza losowe odchylenia zgodnie z rozkładem normalnym w podanym przedziale od *min* do *max*.

Przykład 20. Przyjmijmy jako rozpatrywaną funkcję f funkcję Schaffer-a oraz punkt startowy $x_{start} = (3, 4)$.

Ponadto, przyjmijmy jako paramety metody gradientu prostego oraz metody najszybszego spadku takie same wartości jak w przypadku poprzedniego przykładu z wyjątkiem zmniejszenia maksymalnej iteracji do 2000. Załóżmy również, iż algorytm *simulated annealing* przyjmuje parametry $d = 0.8$, $t_0 = 100$, $\alpha = 0.99$ oraz maksymalny limit iteracji $K = 2000$.

Przy podjęciu próby optymalizacji funkcji z punktu startowego x_{start} przy pomocy algorytmu *simulated annealing* możemy zauważyć, iż eksploruje on rózne regiony badanej funkcji. Nie zatrzymuje sie on w minimach lokalnych a nawet opuszcza ekstermum lokalne w celu badania dalszych wartości. Ścieżki uzyskane w kolejnych krokach iteracji tych algorytmów możemy zaobserwować na wykresie poniżej:

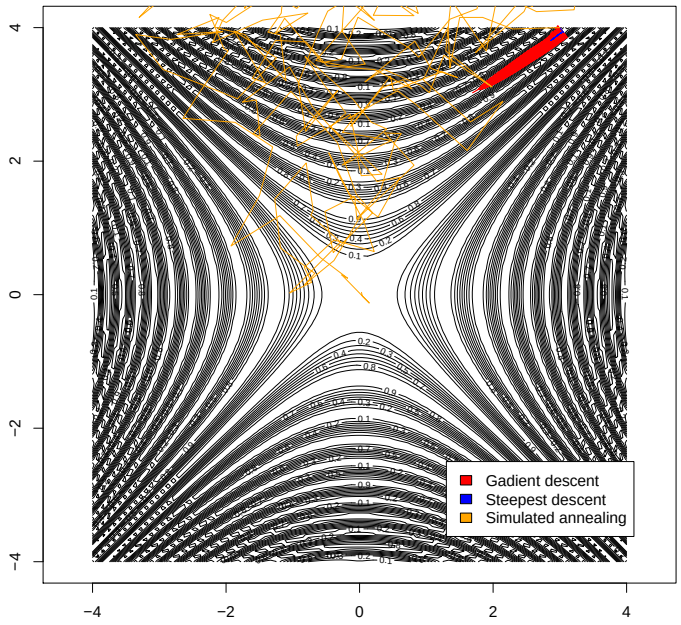


Figure 6.27: Ścieżka stworzona przez przeszukiwanie algorytmem *simulated annealing* w porównaniu do gradient descent oraz steepest descent na bazie funkcji Schaffer-a.

Podczas kolejnych iteracji algorytmu wartość parametru temperatury t_k sukcesywanie spada. Zmniejsza to losowość występującą w zachowaniu algorytmu podczas przeszukiwania i koncentruje go na dotychczas najlepszym znalezionym rozwiązaniu. Algorytm z czasem więc przestaje skupiać się na eksploracji nowych wartości, a stara się zoptymalizować najlepsze dotychczasowe rozwiązanie. Spadek temperatury w kolejnych iteracjach algorytmu możemy zaobserwować na wykresie poniżej:

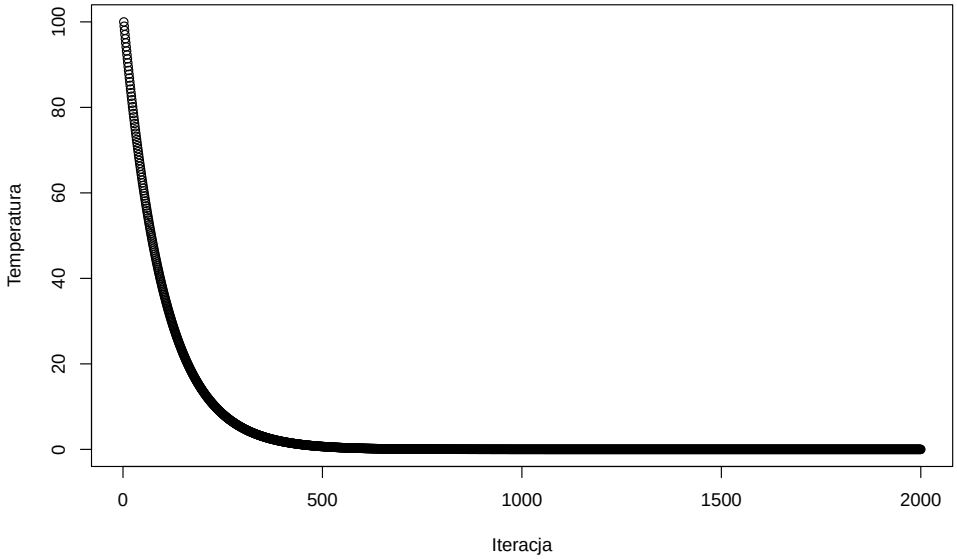


Figure 6.28: Wykres wartości temperatury na przestrzeni kolejnych iteracji algorytmu.

Warto również przyjrzeć się wartościom funkcji aktywacji w kolejnych iteracjach algorytmu. Spadają one na przestrzeni działania algorytmu od wartości bliskich 1 do wartości bliskich 0. Wartości funkcji aktywacji możemy zobaczyć na wykresie poniżej:

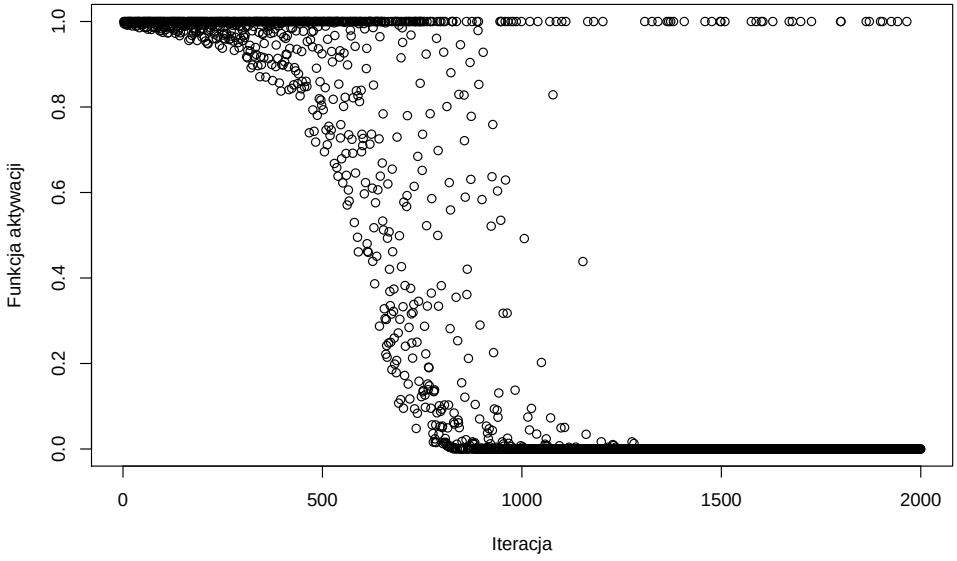


Figure 6.29: Wykres wartości funkcji aktywacji na przestrzeni kolejnych iteracji algorytmu.

Podczas wyszukiwania rozwiązania można wyróżnic trzy fazy, łatwe do zaobserwowania na wykresie funkcji aktywacji. Na początku funkcja aktywacji posiada duże wartości (mniej więcej do 200-300 iteracji). Oznacza to, że algorytm często akceptuje rozpatrywanych kandydatów na optymalny x, nawet jeśli wartości funkcji celu nie są w ich przypadku pożądane. Potem następuje druga faza, w której dobre rozwiązania mają wysokie prawdopodobieństwo, a słabe rozwiązania mają niskie prawdopodobieństwo akceptacji (mniej więcej do 1000-1200 iteracji). Ostatnią fazą jest faza, w której przyjmowanie jest zachowanie modelu przełącznikowego - lepsze rozwiązania przyjmują wartość 1, a gorsze 0. Jedynki występują znacznie rzadziej, gdyż ciężej jest znaleźć lepsze rozwiązanie im dłużej działa algorytm, ponieważ jest wtedy mniejsza szansa na polepszenie wyniku.

Wymienione trzy fazy działania algorytmu odzwierciedlają również wartości funkcji celu, uzyskiwane podczas działania algorytmu. Wykres przedstawiający wspomniane wartości jest umieszczony poniżej:

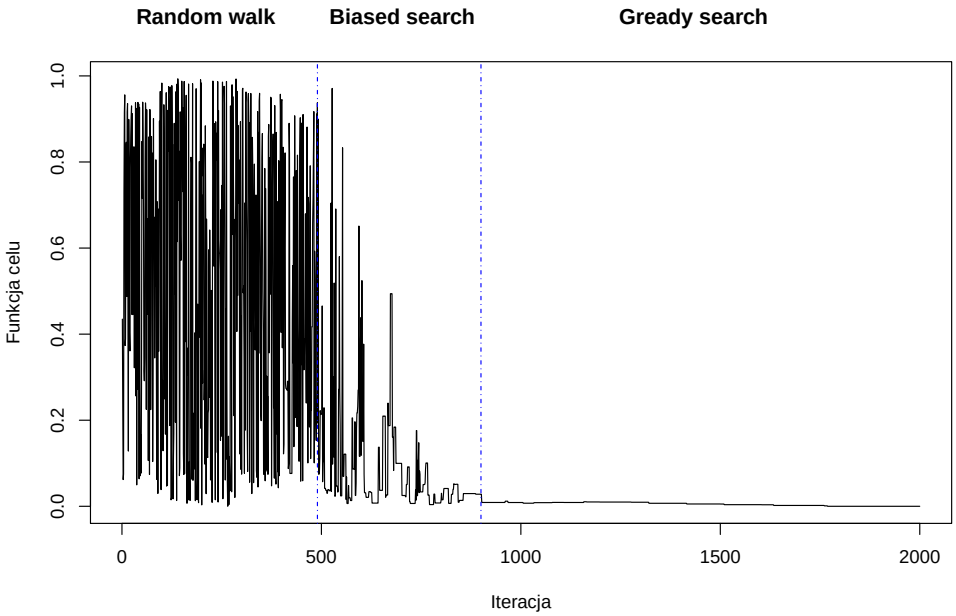


Figure 6.30: Wykres wartości funkcji celu na przestrzeni kolejnych iteracji algorytmu.

Podobnie jak w przypadku funkcji aktywacji, widoczną są trzy fazy działania algorytmu. Na poczatku wartości funkcji celu przyjmują wartości rozrzucone od minimum do maksimum funkcji, gdyż akceptowane były prawie wszystkie napotkane rozwiązania. Następnie częstotliwość i natężenie skoków wartości funkcji zmalała i nie były one już tak drastyczne. W końcowej fazie wartości funkcji celu ustabilizowały się i powoli malały.

Algorytm *simulated annealing* jest bardzo czuły na definicję otoczenia. Poprawność jego działania zależy w znacznym stopniu od poprawnego dobrania parametrów startowych. Ponadto, cierpi on na przekleństwo wymiarowości występujące w problemach optymalizacji. Wraz ze wzrostem wymiarów funkcji, na których pracujemy, znacznie maleje prawdopodobieństwo wybrania prawidłowego kierunku przy analizie losowego sąsiedniego rozwiązania. Zmniejsza to skuteczność algorytmu w przypadku wielowymiarowych problemów.

Simulated Annealing to algorytm stochastyczny, gdyż przy każdym uruchomieniu algorytmu tworzona jest inna ścieżka. Jest to podstawowa **metoda metaheurystyczna**.

6.21 Wstęp do metod populacyjnych i algorytmów genetycznych

Algorytm *simulated annealing* jest dobrym wstępem do algorytmów populacyjnych. Mają one za zadanie zwiększyć skuteczność algorytmów, poprzez wykonanie kilku uruchomień algorytmów z różnymi parametrami startowymi. Większa ilość uruchomień w przypadku algorytmów zawierających losowość zwiększa szansę na uzyskanie poprawnego wyniku. Dodatkowo poszczególne uruchomienia algorytmów mogą współdzielić część informacji, dzięki czemu jeszcze bardziej zwiększą swoje szanse na znalezienie ekstremum funkcji.

Metody populacyjne w zagadnieniach dotyczących optymalizacji to podejście, w którym rozwiązania są reprezentowane jako osobniki w populacji. Osobniki te podlegają procesowi ulepszania, często poprzez procesy genetyczne inspirowane ewolucją biologiczną. Podstawowym celem tych algorytmów jest znalezienie optymalnego rozwiązania w przestrzeni poszukiwań poprzez eksplorację i eksploatację potencjalnych obszarów.

Do najpopularniejszych metod populacyjnych należą:

- Algorytmy genetyczne, które opierają się na mechanizmach dziedziczenia genetycznego, mutacji i krzyżowania,
- Strategie ewolucyjne, które koncentrują się na ewolucji rozwiązań poprzez modyfikację ich strategii, a nie konkretnych genotypów,
- Algorytmy roju (Particle Swarm Optimization), które opierają się na modelu roju, gdzie rozwiązania są reprezentowane przez cząstki, a proces optymalizacji polega na dostosowywaniu ruchu cząstek w przestrzeni poszukiwań w oparciu o ich doświadczenia.

Wszystkie te metody mają wspólny rdzeń w inspiracji procesami ewolucyjnymi, a ich skuteczność zależy od odpowiedniego dostosowania parametrów i reprezentacji problemu. Algorytmy populacyjne są często stosowane do rozwiązywania problemów optymalizacyjnych w różnych dziedzinach, takich jak inżynieria, nauki przyrodnicze, finanse czy sztuczna inteligencja.

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 7: Algorytm sympleksowy

Daniel Kaszyński

7.22 Programowanie liniowe

Programowanie liniowe jest dziedziną optymalizacji umożliwiającą rozwiązywanie najprostszych problemów optymalizacyjnych - problemów liniowych. Jest to metoda uzyskująca optymalne wyniki dla modeli liniowych:

- Liniowych funkcji celu,
- Liniowych ograniczeń równości oraz nierówności.

Jest szeroko stosowany w różnych dziedzinach, takich jak badania operacyjne, ekonomia, inżynieria i nauki o zarządzaniu, do skutecznego rozwiązywania problemów z alokacją, planowaniem i przydziałem zasobów.

Zbiór możliwych rozwiązań jest zbiorem wypukłym (określonym przez skończony zbiór przecinających się półprzestrzeni - każda określona przez nierówność liniową).

Funkcja celu jest funkcją afiniczną zmiennej rzeczywistej określonej na tym wielościanie. Programowanie liniowe umożliwia znalezienie punktu na tym wielościanie, który daje najmniejszą (lub największą) wartość funkcji celu. W programowaniu liniowym wszystkie funkcje celu i ograniczenia są liniowe. Funkcja celu reprezentuje ilość, którą należy zoptymalizować, natomiast ograniczenia reprezentują istniejące ograniczenia lub warunki, które muszą zostać spełnione. Celem jest znalezienie wartości zmiennych decyzyjnych, które optymalizują funkcję celu, spełniając jednocześnie wszystkie ograniczenia.

Ogólna postać problemu programowania liniowego jest następująca:

- Znajdź wektor x
- który maksymalizuje $c^T x$
- pod warunkiem że $Ax \leqslant b$
- oraz $x \geqslant 0$

gdzie wektor x jest wektorem którego składowe mają zostać określone, c jest podanym wektorem wartości, które chcemy zoptymalizować, b jest danym wektorem elementów, do których się ograniczamy oraz A to macierz zmiennych obok ograniczeń. Funkcję $c^T x$, której wartość ma być maksymalizowana, nazywamy funkcją celu. Wypukły wielobok, nad którym ma zostać zoptymalizowana funkcja celu, jest konstruowany przy uwzględnieniu ograniczeń $Ax \leqslant b$ oraz $x \geqslant 0$.

7.23 Algorytm sympleksowy

7.23.1 Podstawy algorytmu

Algorytm sympleksowy jest szeroko stosowaną metodą rozwiązywania problemów programowania liniowego. Jest często uważany za jeden z najskuteczniejszych algorytmów rozwiązywania problemów liniowych

w praktyce.

Algorytm simpleks działa poprzez iteracyjne przechodzenie od jednego wierzchołka (punktu narożnego) dopuszczalnego obszaru do drugiego wzdłuż krawędzi wieloboku określonego przez ograniczenia, aż do osiągnięcia optymalnego rozwiązania. W każdym kroku algorytm wybiera element kluczowy, w celu poprawy wartość funkcji celu i przechodzi do sąsiedniego wierzchołka odpowiadającego elementowi kluczowemu.

Algorytm sympleksowy wykorzystuje standardową formę do reprezentowania nierówności. Nierówności są zatem konwertowane na równości zawierające dodatkowe **zmienne swobodne**.

Przykład 21. Niech $f(x, y, z) = -x + 3y + 2z$ którą chcemy zmaksymalizować pod następującymi warunkami:

$$\begin{cases} x + y + z \leqslant 6 \\ x + z \leqslant 4 \\ y + z \leqslant 3 \\ x + y \leqslant 2 \end{cases}$$

zakładając, że $x, y, z \geqslant 0$. Ograniczające nierówności reprezentowane w postaci standardowej przy użyciu zmiennych swobodnych wyglądałyby następująco:

$$\begin{cases} x + y + z + r + s + t + u = 6 \\ x + z + r + s + t + u = 4 \\ y + z + r + s + t + u = 3 \\ x + y + r + s + t + u = 2 \end{cases}$$

natomiast funkcja f równała by się $-x + 3y + 2z + r + s + t + u$.

7.23.2 Tabela sympleksowa

Algorytm sympleksowy często wykorzystuje reprezentację analizowanego problemu w postaci **tabeli sympleksowej**.

Zakładając, że funkcja f, którą chcemy maksymalizować, jest równa $-x+3y+2z$ i maksymalizujemy ją pod następującymi warunkami:

$$\begin{cases} x + y + z \leqslant 6 \\ x + z \leqslant 4 \\ y + z \leqslant 3 \\ x + y \leqslant 2 \\ x, y, z \geqslant 0 \end{cases}$$

przykładowa tabela sympleksowa dla tego przypadku wyglądałaby tak:

Table 7.1: Przykładowa tabela sympleksowa

x	y	z	r	s	t	u	
1	1	1	1	0	0	0	6
1	0	1	0	1	0	0	4
0	1	1	0	0	1	0	3
1	1	0	0	0	0	1	2
-1	3	2	0	0	0	0	0

W utworzonej tabeli obszar zaznaczony na żółto to macierz A - macierz wartości obok ograniczeń nierówności. W dolnym wierszu widzimy współczynniki funkcji celu. Kolumna po prawej stronie tabeli zawiera wartości, do których ograniczają nierówności - b (wartości po prawej stronie nierówności). Na koniec mamy także 4 zmienne swobodne, po jednej dla każdego ograniczenia. Macierz jednostkowa dla tych zmiennych oznaczona jest w utworzonej tabeli kolorem różowym.

Jeżeli kolumna tabeli odpowiadająca danej zmiennej zawiera same zera i jedną jedynkę, to dana zmienna będzie niezerowa. W przeciwnym razie zmienna ta będzie wynosić zero.

Początkowa tabela simpleksowa jest skonstruowana w taki sposób, że zmienne x, y, z są równe zeru, a zmienne luzu są niezerowe i należą do bazy.

7.23.3 Metoda eliminacji Gaussa

Kolejnym krokiem w algorytmie simpleksowym jest zastosowanie algorytmu eliminacji Gaussa.

Metoda eliminacji Gaussa jest algorytmem stosowanym do rozwiązywania układów równań liniowych poprzez transformację rozszerzonej macierzy reprezentującej układ do postaci schodkowej poprzez serię elementarnych operacji na wierszach. Jest to podstawowa technika algebry liniowej i posiada wiele zastosowań, takich jak rozwiązywanie układów równań liniowych, obliczanie odwrotności macierzy oraz znajdowanie wartości własnych i wektorów własnych. Jest szeroko stosowaną metodą rozwiązywania problemów algebraicznych liniowych, ponieważ jest wydajna i stabilna.

Pierwszym krokiem eliminacji Gaussa jest wybranie kolumny (zmiennej) o największej wartości w dolnym wierszu tabeli. Jeśli kilka zmiennych posiada największą wartość, możemy wybrać dowolną z nich. Wybrana zmienna będzie nową zmienną podstawową wchodzącą do bazy i stanie się *kolumną kluczową*. Kontynuując przykład z podrozdziału o tabeli symplexowej, największy wpływ posiada zmienna y (ponieważ jest mnożona aż przez 3).

Następnie musimy określić, która zmienna swobodna zostanie zastąpiona wybraną zmienną. W tym celu należy obliczyć stosunek kolumny b (kolumny znajdującej się najbardziej na prawo) do wybranej zmiennej. Na szczęście wszystkie wartości kolumny y w naszym przypadku wynoszą 1 lub 0. Powinniśmy wybrać wiersz z **minimalnym stosunkiem**, czyli w tym przypadku ostatnie z ograniczeń. Wiersz z minimalnym stosunkiem nazywa się *wierszem kluczowym*.

Table 7.2: Wiersz i kolumna kluczowa w przykładowej tabeli sympleksowej

x	y	z	r	s	t	u	
1	1	1	1	0	0	0	6
1	0	1	0	1	0	0	4
0	1	1	0	0	1	0	3
1	1	0	0	0	0	1	2
-1	3	2	0	0	0	0	0

Następnym krokiem jest wykonanie szeregu elementarnych operacji na wierszach, tak aby kolumna kluczowa stała się kolumną jednostkową z 1 na przecięciu z wierszem kluczowym (na *elemencie kluczowym*). Aby to osiągnąć, musimy najpierw podzielić wszystkie elementy wiersza kluczowego przez element kluczowy (w naszym przykładzie element kluczowy jest równy 1, więc wiersz kluczowy pozostaje taki sam). Następnie dzielimy pozostałe wiersze przez wiersz kluczowy, aby uzyskać zera w kolumnie kluczowej.

Table 7.3: Przykładowa tabela sympleksowa po pierwszej iteracji algorytmu

x	y	z	r	s	t	u	
0	0	1	1	0	0	-1	4
1	0	1	0	1	0	0	4
-1	0	1	0	0	1	-1	1
1	1	0	0	0	0	1	2
-4	0	2	0	0	0	-3	-6

Te kroki są powtarzane, aż wszystkie wartości w dolnym wierszu będą równe lub mniejsze od zera. Zapewnia to o tym, iż nie ma możliwości dalszej poprawy wartości analizowanego rozwiązania.

Table 7.4: Przykładowa tabela sympleksowa po wszystkich iteracjach algorytmu

x	y	z	r	s	t	u	
1	0	0	1	0	-1	0	3
2	0	0	0	1	-1	1	3
-1	0	1	0	0	1	-1	1
1	1	0	0	0	0	1	2
-2	0	0	0	0	-2	-1	-8

Finalne rozwiązanie w tym przykładzie to $x = 0$, $y = 2$ i $z = 1$. Zmienna x wynosi zero, ponieważ nie jest kolumną jednostkową, natomiast wartości innych zmiennych można odczytać z kolumny znajdującej się najbardziej na prawo.

7.23.4 Implementacja algorytmu sympleksowego

Aby zaimplementować algorytm simpleksowy, musimy najpierw przygotować funkcję odpowiedzialną za tworzenie tabeli sympleksowej. Odczyta ona najpierw liczbę podanych zmiennych, a następnie przygotuje macierz wartości dla początkowej tabeli sympleksowej. Przykładowa implementacja tej funkcji przy użyciu języka programowania **R** mogłaby wyglądać następująco:

```
c_vec <- c(-1, 3, 2)
b_vec <- c(6, 4, 3, 2)
A_mat <- matrix(data = c(1, 1, 1,
                         1, 0, 1,
                         0, 1, 1,
                         1, 1, 0),
                nrow = length(b_vec),
                ncol = length(c_vec),
                byrow = TRUE)

create_simplex_table <- function(A_mat, b_vec, c_vec){
  # Odczytaj liczbe zmiennych/ograniczen
  n_var <- length(c_vec)
  n_con <- length(b_vec)

  # Skonstruuj tabele Simplex
```

```
  st <- matrix(0,
               nrow = n_con+1,
               ncol = n_var+n_con+1)

  st[1:n_con, 1 : n_var] <- A_mat
  st[nrow(st), 1:n_var] <- c_vec
  st[1:n_con, ncol(st)] <- b_vec
  st[1:n_con, (n_var+1): (n_var+n_con)] <- diag(n_con)
  return(st)
}
```

Listing 25: Implementacja funkcji odpowiedzialnej za utworzenie tabeli sympleksowej

Algorytm simpleksowy wymaga również implementacji metody eliminacji Gaussa. Pojedynczy krok tej metody będzie polegał na wybraniu kolumny kluczowej i wiersza kluczowego oraz wykorzystaniu ich do przekształcenia wybranej kolumny w kolumnę jednostkową. Warto zaznaczyć, że w tym celu możemy wykorzystać wbudowaną metodę języka **R** - *outer()*. Umożliwia ona utworzenie nowej macierzy lub tablicy poprzez zastosowanie podanej jako parametr funkcji do każdej możliwej kombinacji elementów z dwóch wektorów wejściowych (gdzie domyślną funkcją jest mnożenie). Implementację algorytmu eliminacji Gaussa można stworzyć w następujący sposób:

```
gaussian_elimination <- function(st, b_vec, c_vec){
  # Odczytaj liczbe zmiennych/ograniczen
  n_var <- length(c_vec)
  n_con <- length(b_vec)

  # Wybierz id kolumny
  i_col <- which.max(st[nrow(st), 1 : n_var])
  if(st[nrow(st), i_col] <= 0){
    return(TRUE)
  }

  # Wybierz id wiersza
  temp <- st[1:n_con,i_col]
  temp <- st[1:n_con,ncol(st)] / temp
  i_row <- which.min(temp)

  # Eliminacja Gaussa
  temp <- st[i_row, ] / st[i_row, i_col]
  st <- st - outer(st[, i_col], st[i_row, ])
  st[i_row, ] <- temp
  return(st)
}
```

Listing 26: Implementacja metody eliminacji Gaussa

Oczywiście na końcu algorytmu chcielibyśmy także móc odczytać wyniki naszych obliczeń. Pomocną może okazać się metoda odczytu wyników w oparciu o uzyskaną tabelę sympleksową. Przykładowa metoda tego typu może wyglądać następująco:

```
read_results <- function(st, b_vec, c_vec){
  # Odczytaj wyniki
  n_var <- length(c_vec)
  n_con <- length(b_vec)

  x_opt <- rep(NA, n_var)
  for(i in 1 : n_var){
    if((sum(st[,i])==1) & (max(st[,i])==1) & (min(st[,i])==0)){
      id <- which(st[,i]==1)
      x_opt[i] <- st[id, ncol(st)]
    }else{
      x_opt[i] <- 0
```

```
    }
  }
  out <- list(x_opt = x_opt,
              f_opt = sum(x_opt * c_vec))
  return(out)
}
```

Listing 27: Implementacja funkcji odczytującej wynik algorytmu

Mając pod ręką wszystkie niezbędne implementacje funkcji, możemy przystąpić do implementacji samego algorytmu. Algorytm simpleksowy powinien składać się z następujących kroków:

- utworzenie początkowej tabeli sympleksowej,
- powtarzanie iteracji metody eliminacji Gaussa, aż wszystkie wartości w dolnym wierszu będą równe lub mniejsze od zera,
- alternatywnie algorytm eliminacji Gaussa powinien zakończyć się po określonej z góry maksymalnej liczbie kroków k,
- odczytanie wyników.

Ostateczna implementacja algorytmu wygląda następująco:

```
simplex_algorithm <- function(A_mat, b_vec, c_vec){
  # Utworz tabele Simplex
  st <- create_simplex_table(A_mat, b_vec, c_vec)

  # Eliminacja Gaussa
  k <- 1
  while(TRUE){
    temp <- gaussian_elimination(st, b_vec, c_vec)
    if((length(temp) == 1) || (k >= 100)){
      break
    }
    st <- temp
    k <- k +1
  }

  # Odczytaj wyniki
  out <- read_results(st, b_vec, c_vec)
  return(out)
}
```

Listing 28: Implementacja algorytmu sympleksowego

131890-S Metody Optymalizacji **Letni 2024/2025**

Zajęcia 8: Optymalizacja dyskretna

Daniel Kaszyński

8.24 Wprowadzenie do optymalizacji dyskretnej

Optymalizacja dyskretna jest działem zajmującym się problemami, gdzie conajmniej jedna ze zmiennych przyjmuje dyskretne wartości. Problemy tego typu są wszechobecne i możemy je napotkać w wielu dziedzinach:

1. **Logistyka** - zarządzanie czasem potrzebnym na rozładunek, załadunek, weryfikację i transport towarów, a także planowanie tras,
2. **Zarządzanie energią** - dostosowanie produkcji energii do zapotrzebowania,
3. **Planowanie rozkładów** - planowanie harmonogramów dla studentów i wykładowców, określanie kto, kiedy, z kim i gdzie,
4. **Rozkłady gier sportowych** - planowanie harmonogramów meczów, w tym kto gra, kiedy, na którym stadionie, biorąc pod uwagę frekwencję widzów i czas transmisji, aby zmaksymalizować zyski.

Problemy optymalizacji są z reguły problemami klasy NP (nondeterministic polynomial, niedeterministycznie wielomianowy). Są to problemy, dla których czas potrzebny na znalezienie optymalnego rozwiązania rośnie wykładniczo wraz z liczbą elementów. W ten sposób możemy szybko znaleźć rozwiązania tylko dla małych problemów, ale są one często zbyt małe, aby zaspokoić nasze potrzeby w świecie rzeczywistym.

Problem ten można zilustrować wykresem pokazanym na Rysunku 8.31. Oś X reprezentuje liczbę elementów dla danego problemu, a oś Y czas potrzebny do znalezienia optymalnego rozwiązania. W idealnym przypadku chcielibyśmy mieć algorytm, który rozwiązuje problem w czasie liniowym. Jednak ze względu na złożoną naturę problemu, liczba możliwych rozwiązań rośnie nieliniowo.

W problemach typu NP, czesto jesteśmy w stanie stwierdzić czy proponowane rozwiązanie jest poprawne na pierwszy rzut oka. Ciężko jest natomiast znaleźć lub określić postać rozwiązania optymalnego ze względu na duża liczbę możliwych rozwiązań. Z tego powodu podchodząc na problemów dyskretnych często podchodzimy do rozwiązywania ich na dwa sposoby:

1. **Przesunięcie linii wykładniczego wzrostu** - dostosowanie wzrostu potrzebnego czasu, abyśmy mogli rozwiązywać większe problemy, zanim czas rozwiązania stanie się znaczącym problemem,
2. **Znalezienie przybliżonych rozwiązań** - poszukiwanie rozwiązań, które nie są optymalne, ale zapewniają satysfakcjonujące wyniki w znacznie krótszym czasie.

Aby dokładniej zbadać problemy optymalizacji dyskretnej, przyjrzyjmy się bliżej konkretnym problemom optymalizacyjnym, badając ich złożoność i możliwe podejścia do ich rozwiązywania.

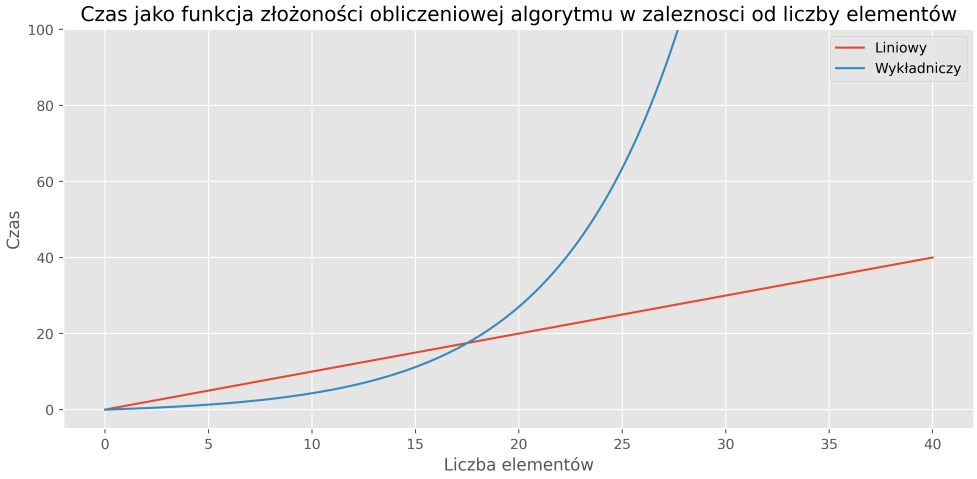


Figure 8.31: Czas wymagany do rozwiązania problemu w zależności od liczby elementów i złożoności obliczeniowej

8.25 Problem plecakowy

Wyobraźmy sobie sytuację, w której złodziej włamuje się do muzeum z zamiarem kradzieży dzieł sztuki i sprzedania ich. Złodziej ma ze sobą plecak, ale ma on ograniczoną pojemność (dla uproszczenia zignorujmy wymiary przedmiotów), co oznacza, że złodziej nie może zabrać wszystkich dzieł sztuki. Dlatego złodziej musi zdecydować, które przedmioty wybrać, aby zabrać najcenniejsze dzieła sztuki, które zmieszczą się w plecaku i przyniosą największy zysk.

Sformalizujmy problem matematycznie: mamy zbiór wszystkich przedmiotów w muzeum $\mathcal{I} = (i_1, i_2, \ldots, i_n)$. Każdy przedmiot ma swoją wartość i wagę $i_i = (v_i, w_i)$. Mamy również maksymalną pojemność plecaka K oraz dodatkowe zmienne decyzyjne $x = (x_1, x_2, \ldots, x_n)$, gdzie $x_i = 1$ jeżeli dany przedmiot umieszczamy w plecaku oraz $x_i = 0$ w przeciwnym wypadku. Możemy teraz przedstawić problem optymalizacyjny jako zadanie mając na celu maksymalizacje wartości plecaka pod warunkiem ograniczenia w postaci maksymalnej pojemności plecaka:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{N} w_i x_i \leqslant K \\ & x_i \in \{0,1\}, \quad \forall i \in \{1,2,\ldots,N\} \end{aligned} \tag{8.59}$$

Jest to klasyczny przykład **problemu plecakowego** podstawowego problemu kombinatorycznego w optymalizacji. Celem jest zebranie najcenniejszej kombinacji przedmiotów, która zmieści się w plecaku.

Zanim przejdziemy do omówienia sposobów rozwiązania tego problemu, warto zastanowić się, ile czasu zajęłoby sprawdzenie wszystkich możliwych rozwiązań, innymi słowy przeprowadzenie dokładnego wyszukiwania. Zmienna decyzyjna x może przyjmować następujące wartości: $(0, 0, \ldots, 0)$, $(0, 0, \ldots, 1)$, $\ldots$, $(1, 1, \ldots, 1)$. Liczba wszystkich możliwych kombinacji jest związana z liczbą elementów w zbiorze $\mathcal{I}$, a w tym przypadku

jest równa $2^{|\mathcal{I}|}$. Zakładając, że sprawdzenie pojedynczego przypadku zajmuje jedną milisekundę, dla 50-elementowego zbioru $|\mathcal{I}| = 50$ sprawdzenie wszystkich możliwych przypadków zajęłoby 35677 lat. Dlatego widzimy, że takie podejście musi zostać wykluczone, ponieważ jest praktycznie niewykonalne.

W kolejnych sekcjach rozważymy podejścia, które wykorzystują metody takie jak **greedy search**, **dynamic programming** i **branch and bound** do projektowania dedykowanych algorytmów optymalizacyjnych - heurystyk, które znajdują wykonalne rozwiązania w znacznie krótszym czasie.

8.26 Greedy search

Algorytmy Greedy search są dobrymi strategiami, aby znaleźć początkowe prawidłowe rozwiązania spełniające ograniczenia problemu optymalizacyjnego. Algorytmy te budują rozwiązania w konceptualnie prosty i intuicyjny sposób. Ogólnie rzecz biorąc, takie algorytmy mogą nie być najbardziej wydajne, ale posłużą jako pierwsze kroki w kierunku głębszego zrozumienia problemu, co może być później przydatne przy stosowaniu bardziej wyrafinowanych metod optymalizacji. Przyjrzyjmy się problemowi opisanemu równaniem (8.60):

$$\begin{aligned} & x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \\ & 1x_1 + 1x_2 + 1x_3 + 10x_4 + 11x_5 + 13x_6 + 7x_7 \\ & 2x_1 + 2x_2 + 2x_3 + 5x_4 + 5x_5 + 8x_6 + 3x_7 \leqslant 10 \end{aligned} \tag{8.60}$$

Mamy do dyspozycji 7 przedmiotów, pierwsze równanie opisuje przypisanie przedmiotów do plecaka, drugie jest funkcją celu z przypisanymi wartościami przedmiotów w dolarach (1 dolar, 1 dolar, ...), a nierówność zawiera wagi tych przedmiotów w kg (2kg, 2kg, ...) wraz z maksymalnym udźwigiem plecaka $K = 10$ kg. Potencjalną strategią rozwiązania tego problemu może być początkowe wybranie najcięższych przedmiotów, a następnie wypełnienie plecaka tak, aby nie przekroczyć maksymalnego obciążenia:

$$\begin{aligned} & x = (1, 0, 0, 0, 0, 1, 0) \\ & 1 * 1 + 1 * 0 + 1 * 0 + 10 * 0 + 10 * 0 + 13 * 1 + 7 * 0 = 14 \\ & 2 * 1 + 2 * 0 + 2 * 0 + 5 * 0 + 5 * 0 + 8 * 1 + 3 * 0 = 10 \leqslant 10 \end{aligned} \tag{8.61}$$

Wypróbujmy kilka innych strategii. Zastosujmy podejście „im więcej, tym lepiej", wypełniając najpierw plecak najlżejszymi przedmiotami:

$$\begin{aligned} & x = (1, 1, 1, 0, 0, 0, 1) \\ & 1 * 1 + 1 * 1 + 1 * 1 + 10 * 0 + 10 * 0 + 13 * 0 + 7 * 1 = 10 \\ & 2 * 1 + 2 * 1 + 2 * 1 + 5 * 0 + 5 * 0 + 8 * 0 + 3 * 1 = 9 \leqslant 10 \end{aligned} \tag{8.62}$$

Zmiana strategii spowodowała, że nie wykorzystano w pełni dostępnego miejsca w plecaku i jego wartość spadła o 4 dolary.

Kolejna strategia, którą przyjmiemy, wymaga dodatkowych obliczeń: wybieramy przedmioty o najwyższej wartości na kilogram wagi, co oznacza, że dla przedmiotu x_1 mamy $\frac{warto}{waga} = \frac{1}{2}$. Utwórzmy tymczasową listę z pozycjami efektywność kosztowa (CE), której użyjemy do sterowania wyborem przedmiotów:

$$CE = \left[\frac{1}{2},\ \frac{1}{2},\ \frac{1}{2},\ 2,\ 2,\ 1\frac{5}{8},\ 2\frac{1}{3}\right] \tag{8.63}$$

Wybierając elementy na podstawie ich najwyższych wartości z listy CE, będziemy wybierać kolejno $(x_7, x_4, x_5, x_6, x_1, x_2, x_3)$. Musimy jednak wziąć pod uwagę maksymalną wagę plecaka, ponieważ wybór (x_7, x_4, x_5) przekroczy limit 10 kg:

$$\begin{aligned} &x = (1, 0, 0, 1, 0, 0, 1) \\ &1 * 1 + 1 * 0 + 1 * 0 + 10 * 1 + 10 * 0 + 13 * 0 + 7 * 1 = 18 \\ &2 * 1 + 2 * 0 + 2 * 0 + 5 * 1 + 5 * 0 + 8 * 0 + 3 * 1 = 10 \leqslant 10 \end{aligned} \tag{8.64}$$

Udało nam się opracować jeszcze lepszą strategię, ale pozostaje pytanie, czy jest to najlepsza strategia. Łatwo zauważyć, że wybierając przedmioty x_4 i x_5, wartość plecaka wynosi 20 dolarów. Jak możemy opracować algorytm, który pozwoli nam osiągnąć ten wynik? Powinniśmy też zadać sobie pytanie czy jest to naprawdę optymalne rozwiązanie?

Podsumujmy algorytmy zachłanne. Wiemy, że możemy zaproponować wiele różnych algorytmów do rozwiązania danego problemu, z których niektóre są mniej lub bardziej efektywne od innych. Algorytmy zachłanne są proste w koncepcji i implementacji i generalnie działają bardzo szybko. Mają jednak pewne ograniczenia - nie zawsze znajdują optymalne rozwiązanie, a ich skuteczność może się różnić w zależności od danych wejściowych. Pomimo tych ograniczeń, algorytmy zachłanne są nadal przydatne do tworzenia benchmarków i porównywania wydajności innych metod optymalizacji dla danego problemu.

8.26.1 Dynamic programming

Programowanie dynamiczne jest techniką rozwiązywania problemów, która wykorzystuje podejście *divide and conquer* oraz *bottom-up computation*. W podejściu divide and conquer, dzielimy problem na mniejsze podproblemy i używamy ich do znalezienia optymalnego rozwiązania. Oznaczmy optymalne rozwiązanie przez $O(k, j)$, gdzie $k \in [0, K]$ reprezentuje optymalną pojemność plecaka, a $j \in [0, n]$ reprezentuje optymalną liczbę przedmiotów w plecaku. W tym podejściu, zamiast rozwiązywać problem dla wszystkich N przedmiotów na raz, rozwiązujemy go dla każdego kolejnego j.

Załóżmy, że wiemy jak rozwiązać $O(k, j-1)$ dla wszystkich $k \in [0, K]$ i chcemy dowiedzieć się jak rozwiązać $O(k, j)$, co oznacza dodanie przedmiotu j do plecaka:

1. Jeśli waga następnego przedmiotu w_j jest większa niż k-te maksymalne obciążenie plecaka, to najlepszym rozwiązaniem jest $O(k, j-1)$. Dzieje się tak, ponieważ przedmiot j nie mieści się do aktualnej pojemności.
2. W drugim przypadku, gdy $w_j \leqslant k$, możemy rozważyć dwa scenariusze i wybrać ten, który daje lepszy wynik: (1) nie wybieraj j-tego elementu, najlepszym rozwiązaniem jest $O(k, j-1)$. (2) wybierz j-tego elementu, rozwiązaniem jest wartość elementu j i optymalna wartość poprzednio rozważanego plecaka ze zmniejszoną pojemnością o element j: $v_j + O(k - w_j, j-1)$.

Zapiszmy algorytm w formie rownania (8.65):

$$O(k, j) = \begin{cases} max(O(k, j-1), v_j + O(k - w_j, j-1)) & \text{if } w_j \leqslant k \\ O(k, j-1) & \text{else} \end{cases} \tag{8.65}$$

Jak widać, jest to równanie rekurencyjne, które można łatwo zaimplementować w R, jak pokazano na listingu kodu 29. Należy zauważyć, że problemy rekurencyjne wymagają przypadku bazowego, który pozwala nam zatrzymać obliczenia, jeśli nie możemy dalej zredukować problemu. Przypadek bazowy jest tutaj określony dla *item_idx==0*, co oznacza, że nie dodajemy żadnych przedmiotów do plecaka (nie ma on wartości).

```r
N <- 3  # maksymalna liczba przedmiotow
K <- 9  # maksymalna ladownosc plecaka
# wartosc i waga kolejnych przedmiotow
item_values <- c(5, 6, 3)
item_weights <- c(4, 5, 2)

O_ALG <- function(capacity, item_idx) {
  # 0 jako index brak przedmiotu -> brak wartosci
  if (item_idx == 0) {
    return(0)
  }

  # wyciagniecie informacji przedmiotu
  value <- item_values[item_idx]
  weight <- item_weights[item_idx]

  # czy mozemy dodac przedmiot do plecaka
  if (weight <= capacity) {
    # Wybieramy maksimum z
    # 1. przypadek gdzie nei dodajemy przedmiotu do plecaka
    # 2. przypadek gdzie dodajemy przedmiot do plecaka
    return(max(
      O_ALG(capacity, item_idx - 1),
      value + O_ALG(capacity - weight, item_idx - 1)
    ))
  }
  # jest nie ma miejsca na przedmiot to od razu rozwazamy kolejny
  return(O_ALG(capacity, item_idx - 1))
}

cat(sprintf("Optymalna wartosc plecaka: %d\n", O_ALG(K, N)))
```

Listing 29: Problem plecakowy podejście rekurencyjne

Przyjęcie takiego algorytmu rekurencyjnego jest podejściem nieefektywnym obliczeniowo. Osoby zaznajomione z rekurencyjną implementacją znajdowania n-tego elementu ciągu Fibonacciego wiedzą, że problem polega na wielokrotnym obliczaniu tych samych wartości. Metoda ta znana jest jako podejście „od góry do dołu". Aby poprawić wydajność algorytmu, musimy zmienić kierunek wykonywania i przyjąć podejście *top to bottom*. Zamiast zaczynać od j-tego elementu w dół, zaczniemy od elementu zerowego (brak elementów w plecaku) i będziemy zwiększać liczbę elementów o jeden, aż dojdziemy do N-tego elementu. Rozważmy to na prostym przykładzie:

$$\begin{aligned} \max \quad & 5x_1 + 6x_2 + 3x_3 \\ \text{s.t.} \quad & 4x_1 + 5x_2 + 2x_3 \leqslant 9 \end{aligned} \tag{8.66}$$

W programowaniu dynamicznym używamy tabeli, która zawiera informacje o optymalnych wartościach plecaka dla wszystkich możliwych konfiguracji k i j. Tabela ta ma wiersze dla K $w[0, K]$ pojemności i kolumny dla j $w[0, N]$ elementów. Proces budowania kolejnych kolumn tabeli pokazano na rysunku 8.32. Pierwsza kolumna jest początkowo wypełniona zerami, ponieważ jeśli w plecaku nie ma żadnych przedmiotów, jej wartość wynosi zero.

W drugiej kolumnie rozważamy umieszczenie pierwszego przedmiotu w plecaku. Korzystając z reguły decyzyjnej opisanej równaniem (8.65), wiemy, że dla $k < 4$ przyjmujemy wartości z poprzedniej kolumny,

ponieważ nie mamy miejsca w mniejszym plecaku. Dla $k \geqslant 4$ rozważamy równania w postaci $max(O(k,0), 5+ O(k-4,0))$. W tym przypadku sytuacja jest prosta, ponieważ dla każdego k bierzemy $5 + O(k-4,0)$, więc wypełnienie tej kolumny jest proste. Zauważmy, że nie musimy już obliczać $O(k,0)$ i $O(k-4,0)$, ponieważ mamy ich wartości zapisane w tabeli.

Pojemność plecaka	Liczba przedmiotow 0	1	2	3
0	0			
1	0			
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			
8	0			
9	0			

v1 = 5 v2 = 6 v3 = 3
w2 = 4 w2 = 5 w3 = 2

Pojemność plecaka	Liczba przedmiotow 0	1	2	3
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	5		
5	0	5		
6	0	5		
7	0	5		
8	0	5		
9	0	5		

v1 = 5 v2 = 6 v3 = 3
w2 = 4 w2 = 5 w3 = 2

Pojemność plecaka	Liczba przedmiotow 0	1	2	3
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	5	5	
5	0	5	6	
6	0	5	6	
7	0	5	6	
8	0	5	6	
9	0	5	6+5	

v1 = 5 v2 = 6 v3 = 3
w2 = 4 w2 = 5 w3 = 2

Pojemność plecaka	Liczba przedmiotow 0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	5+3
7	0	5	6	6+3
8	0	5	6	6+3
9	0	5	6+5	**6+5**

v1 = 5 v2 = 6 v3 = 3
w2 = 4 w2 = 5 w3 = 2

Figure 8.32: Dynamic programming tabela

Interesujące rzeczy dzieją się, gdy zaczynamy wypełniać trzecią kolumnę. Ponieważ drugi element ma wagę $w_2 = 5$, dla plecaka o pojemności $k = 4$, efektywny plecak to $O(4,1)$. Ale dla $k = 5$, rozważamy $max(O(5,1), 6 + O(5-5,1)) = max(5,6) = 6$. Co więcej, dla $k = 9$, równanie staje się $max(O(9,1), 6 + O(9-5,1)) = max(5, 6+5) = 11$, co jest obecnie najwyższą wartością plecaka.

Musimy powtórzyć procedurę jeszcze raz dla ostatniej kolumny, aby zebrać wszystkie niezbędne informacje. Maksymalna wartość, jaką możemy osiągnąć w plecaku to 11. Co ciekawe, najwyższa wartość zawsze będzie znajdować się w prawym dolnym rogu tabeli.

Pozostaje pytanie, które przedmioty należy umieścić w plecaku, aby osiągnąć daną wartość. W tym przypadku wiemy, że są to i_1 i i_2, ponieważ zaobserwowaliśmy to podczas ręcznego budowania tabeli. Możemy to jednak łatwo zweryfikować po zbudowaniu tabeli bez przechowywania wszystkich informacji po drodze.

Przyjrzyjmy się bliżej rysunkowi 8.66, który pokazuje pełną tabelę z zsumowanymi wartościami. Aby określić, które przedmioty należy umieścić w plecaku, musimy prześledzić drogę od optymalnego koszyka do pustego. Zaczynając od pozycji $O(9,3)$, sprawdzamy pozycję po lewej stronie, $O(9,2)$. Jeśli wartość nie uległa zmianie, oznacza to, że nie umieściliśmy przedmiotu j w plecaku. Usunięcie go z rozważań nie zmienia optymalnej wartości, ponieważ nie było go w plecaku, więc przedmiot i_3 nie znajduje się w optymalnym plecaku. Następnie porównujemy pozycję $O(9,2)$ z $O(9,1)$, w tym przypadku wartość zmieniła się, wskazując, że usunięcie przedmiotu i_2 wpływa na optymalną wartość plecaka, więc wiemy, że musi on znajdować się w plecaku. W

następnym kroku nie zaczynamy od $O(9,1)$, ale od $O(9-5,1)$, co uwzględnia wagę przedmiotu i_2. Porównanie $O(4,1)$ z $O(4,0)$ mówi nam, że przedmiot i_1 został umieszczony w plecaku, osiągając optymalną wartość. Zatem optymalny plecak zawiera przedmioty i_1 i i_2.

Pojemność plecaka	Liczba przedmiotow 0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	9
8	0	5	6	9
9	0	5	11	11
		v1 = 5	v2 = 6	v3 = 3
		w2 = 4	w2 = 5	w3 = 2

Figure 8.33: Dynamic programming tabela - traceback

Listing kodu 30 przedstawia implementację algorytmu dynamic programming dla problemu plecakowego i znajdowania optymalnej wartości i zawartości plecaka. Należy zauważyć, że algorytm ten nie jest pozbawiony wad. W porównaniu do implementacji rekurencyjnej, rezerwuje on miejsce w pamięci dla tabeli ze wszystkimi możliwymi rozwiązaniami, co może stanowić znaczący problem przy dużej liczbie przedmiotów i wysokiej maksymalnej wadze plecaka.

```r
N <- 3  # maksymalna liczba przedmiotow
K <- 9  # maksymalna ladownosc plecaka
# wartosc i waga kolejnych przedmiotow
item_values <- c(5, 6, 3)
item_weights <- c(4, 5, 2)


get_dp_table <- function() {
  # Builds a DP table and fills the entries
  # budowa tabeli DP i uzupelnienie pol
  dp_table <- matrix(0, nrow = K + 1, ncol = N + 1)

  for (item in 1:N) {
    value <- item_values[item]
    weight <- item_weights[item]
    # zaczynajac od pierwszego przedmiotu czy mozemy go dodac?
    for (capacity in 0:K) {
      # check this condition for each possible capacity configuration
      # sprawdzenie tego warunku dla kazdej mozliwej konfiguracji pojemnosci plecaka
      if (weight <= capacity) {
        # jesli mozliwe dodaj i ustal maksimum z
        # 1. nie dodajemy przedmiotu
        # 2. dodajemy przedmiotu do plecaka
        dp_table[capacity + 1, item + 1] <- max(
          dp_table[capacity + 1, item],
          value + dp_table[capacity - weight + 1, item]
        )
      } else {
        # jesli nie mozna dodac przedmiotu to rozwiazanie ostatniego przedmiotu
        dp_table[capacity + 1, item + 1] <- dp_table[capacity + 1, item]
      }
    }
  }

  return(dp_table)
}
```

```r

# wyciagniecie optymalnego rozwiazania
solution_values <- function(dp_table) {
  current_item <- N
  current_capacity <- K
  items_idx <- c()
  total_value <- 0
  total_weight <- 0

  while (current_item != 0) {
    # the item doesnt increase the value function move to the next item
    # jesli dodanie przedmiotu do plecaka nie poprawia rozwiazania to pomijamy przedmiotu
    if (dp_table[current_capacity + 1, current_item + 1] != dp_table[current_capacity + 1,
    current_item]) {
      # zapisanie informacji o przedmiocie dodanym do plecaka
      items_idx <- c(items_idx, current_item)
      value <- item_values[current_item]
      weight <- item_weights[current_item]
      total_value <- total_value + value
      total_weight <- total_weight + weight

      # zredukowanie pozostalego miejsca w plecaki
      current_capacity <- current_capacity - weight
    }
    # przejscie do kolejnego przedmiotu
    current_item <- current_item - 1
  }

  opt_val <- dp_table[K + 1, N + 1]
  return(list(opt_val = opt_val, items_idx = sort(items_idx), total_value = total_value,
    total_weight = total_weight))
}

table <- get_dp_table()
solution_info <- solution_values(table)
print(solution_info)
```

Listing 30: Dynamic programing approach to knapack problem

8.26.2 Branch and bound

Branch and bound to metoda rozwiązywania problemów optymalizacyjnych poprzez konstruowanie drzewa decyzyjnego. Przykład drzewa decyzyjnego dla plecaka z trzema elementami pokazano na rysunku 8.34. Stan na samym szczycie drzewa, zwany korzeniem, reprezentuje pusty plecak. Lewa gałąź drzewa uwzględnia scenariusze z pierwszym przedmiotem w plecaku, a prawa gałąź bez niego. Dla tych dwóch węzłów rozważamy następnie dodanie drugiego przedmiotu do plecaka, co skutkuje czterema nowymi gałęziami. Następnie rozważamy dodanie trzeciego przedmiotu, tworząc w sumie $2^4 - 1 = 15$ stanów reprezentujących wszystkie kombinacje wraz ze stanami pośrednimi.

Gdy liczba elementów $|\mathcal{I}|$ jest zbyt duża, podejście to staje się niepraktyczne, ponieważ niemożliwe byłoby sprawdzenie wszystkich możliwych scenariuszy. Dlatego też musimy znaleźć metodę generowania drzew w taki sposób, aby nie przeszukiwać całego drzewa, ale podejmować decyzje w poszczególnych węzłach w celu określenia, czy jest ono warte dalszej eksploracji. Podzielmy ten problem na dwa etapy:

1. **branching** - rozszerzenie węzła drzewa na dwa węzły,

2. **bounding** - optymistyczna ocena poprzez *relaksacja* ograniczeń problemu optymalizacji.

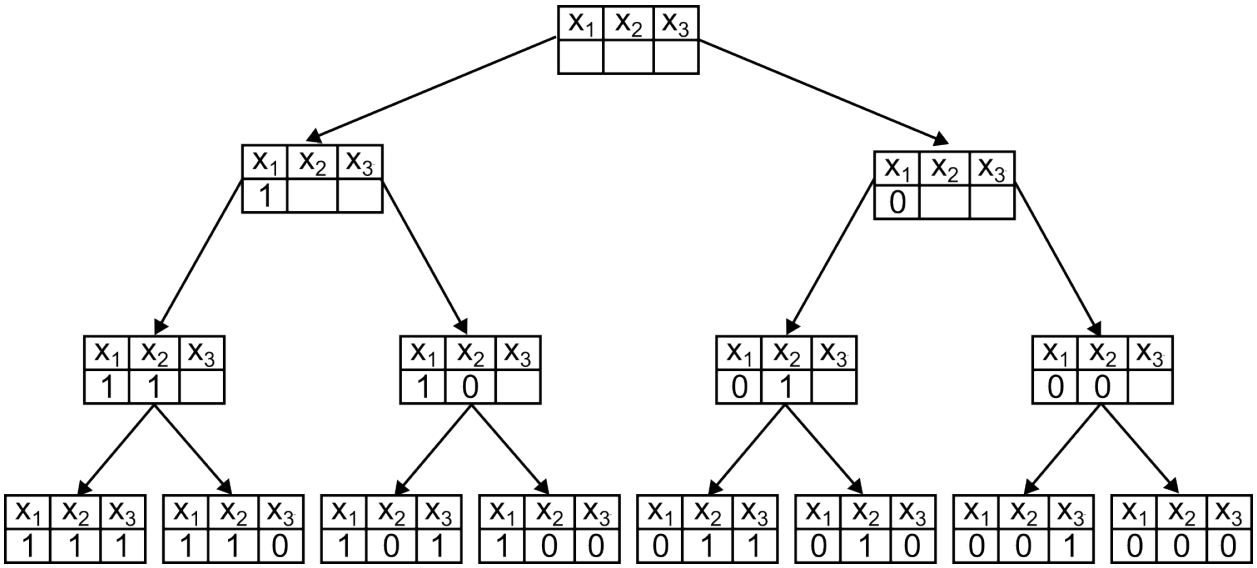


Figure 8.34: Drzewo decyzyjnie dla problemu plecakowego

Aby zrozumieć koncepcje optymistycznej oceny i relaksacji, rozważmy prosty przykład. Mamy trzy przedmioty $i_1 = (v = 45, w = 5)$, $i_2 = (v = 48, w = 8)$ i $i_3 = (v = 35, w = 3)$ z $K = 10$, co oznacza, że nigdy nie zmieścimy wszystkich przedmiotów do plecaka, ale możemy wziąć sam i_2 lub i_1 i i_3. Usuwając ograniczenie maksymalnego obciążenia, naszą optymistyczną oceną jest wartość $45+48+35 = 128$. Możemy teraz rozpocząć przeszukiwanie drzewa, pamiętając o przyszłej optymistycznej wartości dla każdego węzła, porównując i odrzucając nieopłacalne gałęzie.

Rozważmy logikę takiego algorytmu na przykładzie. Rysunek 8.35 pokazuje pierwsze dwa etapy rozgałęziania drzewa. W korzeniu określamy bieżące stany, tj. wartość plecaka jako 0 (brak przedmiotów), pozostałą wolną przestrzeń i optymistyczną wartość plecaka w bieżącym stanie. Rozszerzamy korzeń na dwa węzły: w jednym element i_1 trafia do plecaka, a w drugim odrzucamy go i aktualizujemy stany węzłów. Optymistyczna wartość węzłów różni się, ponieważ prawe węzły $x = (0, ?, ?)$, nie uwzględniają już i_1 w optymistycznej ocenie, więc $48 + 35 = 83$ w najlepszym przypadku.

Rozwińmy dalej węzeł $x = (1, ?, ?)$. Widzimy, że w węźle $x = (1, 1, ?)$ przekraczamy maksymalne obciążenie plecaka, więc możemy natychmiast odrzucić dalszą eksplorację tej ścieżki, ponieważ nigdy nie przyniesie ona poprawnego rozwiązania. W przypadku $x = (1, 0, ?)$ wartość oczekiwana spada do 80, ale daje poprawne rozwiązanie. Dalsze działania mogą się różnić w zależności od przyjętego algorytmu przeszukiwania drzewa, ale na razie kontynuujmy eksplorację węzła $x = (1, 0, ?)$.

Dalsza eksploracja drzewa jest pokazana na rysunku 8.36. Po rozwinięciu węzła $x = (1, 0, ?)$ otrzymujemy przypadek $x = (1, 0, 1)$, gdzie wartość plecaka jest najwyższa i równa optymistycznej wartości 80. Ale otrzymujemy też nieco gorsze rozwiązanie, $x = (1, 0, 0)$ z optymalną wartością 45.

Ponieważ w podejściu relaksacyjnym śledzimy najwyższą optymistyczną wartość, podczas przeglądania drzewa widzimy, że węzeł $x = (0, ?, ?)$ nadal potencjalnie oferuje wyższą optymistyczną wartość niż obecne najlepsze rozwiązanie (80 vs 83). Ponieważ może to być potencjalnie lepsze rozwiązanie, algorytm kontynuuje przeszukiwanie drzewa. Możemy odrzucić węzeł $x = (0, 0, ?)$, ponieważ w porównaniu z optymalnym rozwiązaniem jest on gorszy i nie przyniesie lepszego wyniku. Rozszerzenie węzła $x = (0, 1, ?)$ ujawnia inne optymalne rozwiązanie, $x = (0, 1, 0)$, które jest gorsze od obecnego najlepszego, oraz nieprawidłowe rozwiązanie $x = (0, 1, 1)$, ponieważ przekracza ono pojemność plecaka.

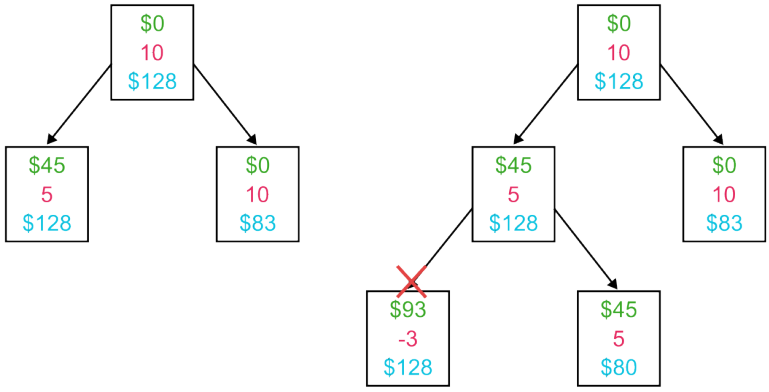


Figure 8.35: Rozszerzanie drzewa 1

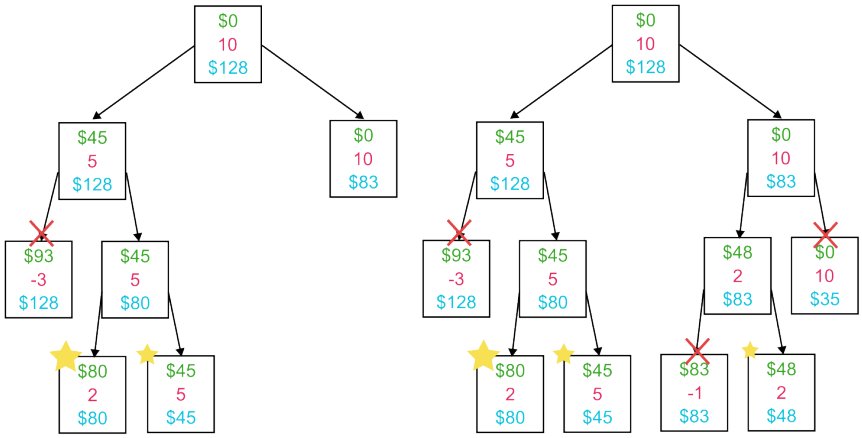


Figure 8.36: Rozwijanie drzewa 2

Widzimy więc, że zastosowanie relaksacji i wartości optymistycznej pozwoliło nam zmniejszyć rozmiar drzewa. Wynik ten może być jednak niezadowalający, ponieważ zmniejszyliśmy drzewo tylko o dwa węzły. W związku z tym możemy znaleźć lepszą metodę relaksacji, która może być kluczowa dla problemów generujących duże drzewa.

Interesującym przykładem takiej reguły może być przełamanie dyskretnej natury problemu poprzez dopuszczenie ułamkowych części przedmiotów do umieszczenia w plecaku. Możemy uwzględnić koncepcję z wcześniej przedstawionego algorytmu zachłannego, aby sortować przedmioty według ich koszt efektywności $frac$valueweight, a w optymistycznej wartości wypełniamy plecak ułamkiem przedmiotu przekraczającym obciążenie. W naszym przykładzie zmieni to początkową wartość optymistyczną na:

$$
\begin{aligned}
&CE = \left[\frac{45}{5},\ \frac{48}{8},\ \frac{35}{3}\right] = \left[9,\ 6,\ 11\frac{2}{3}\right] \\
&w_3 + w_1 + \frac{1}{4}w_2 = 10 \leqslant 10 \\
&OptimisticValue = v_3 + v_1 + \frac{1}{4}v_2 = 35 + 45 + 12 = 92
\end{aligned}
\tag{8.67}
$$

Innymi słowy, dokonujemy reparametryzacji problemu poprzez wprowadzenie $x_i = \frac{y_i}{v_i}$, co skutkuje następującą zmianą równań problemu optymalizacyjnego:

$$
\begin{aligned}
\max \quad & \sum_{\mathcal{I}} y_i \\
\text{s.t.} \quad & \sum_{\mathcal{I}} \frac{w_i}{v_i} y_i \leqslant K \\
& y_i \in [0, 1]
\end{aligned}
\tag{8.68}
$$

Takie podejście oznacza, że po rozwinięciu całego węzła $x = (1, ?, ?)$ i powrocie do $x = (0, ?, ?)$, nie będziemy kontynuować rozwijania drzewa, ponieważ optymistyczna wartość po wykluczeniu i_1 wynosi $35 + \frac{7}{8} \cdot 48 = 77$ ($\frac{7}{8}$, ponieważ najpierw umieszczamy element i_3 z wagą 3, a resztę wypełniamy i_2 z wagą 8).

Oprócz znaczenia metody relaksacji w wyszukiwaniu drzew, możemy również omówić podejścia związane z przechodzeniem przez drzewa. Popularne podejścia obejmują **Depth-first search** i **Best-first search**. Podejście oparte na wyszukiwaniu w głąb to metoda, którą omówiliśmy we wcześniejszym przykładzie. W tym podejściu najpierw skupiamy się na lewej (lub prawej) gałęzi drzewa. Gdy dotrzemy do końca gałęzi, sprawdzamy pozostałe gałęzie. W najlepszym przypadku możemy zakończyć wyszukiwanie po rozwinięciu tylko jednej gałęzi.

Wyszukiwanie Best-first działa zupełnie inaczej. W tym algorytmie rozwijamy węzły o aktualnie najwyższej optymistycznej wartości. Nie ma jednak prostej odpowiedzi na pytanie, które podejście jest lepsze, a czas wymagany do rozwiązania problemu może się różnić w zależności od problemu. Best-first search niesie ze sobą dodatkowe ryzyko w sytuacjach, gdy wartości kilku elementów są nieskończenie duże. W takich przypadkach możemy przeskakiwać od węzła do węzła, prawie odtwarzając całe drzewo, podczas gdy w podejściu wgłębnym możemy zakończyć wyszukiwanie po przejściu jednej lub więcej gałęzi.

Listing 31 przedstawia implementację algorytmu branch and bound wykorzystującego Depth-first search z prostą relaksacją, która usuwa ograniczenie maksymalnego obciążenia plecaka.

```r
N <- 3  # maksymalna liczba przedmiotow
K <- 9  # maksymalna ladownosc plecaka
# wartosc i waga kolejnych przedmiotow
item_values <- c(45, 48, 35)
item_weights <- c(5, 8, 3)

# definicja wezla w postaci listy
Node <- function(level, value, capacity) {
  return(list(
    level = level,
    value = value,
    capacity = capacity,
    items = c(),
    opt_estimate = 0
  ))
}
```

```
17
18 # utility do wyznaczania optymistycznej wartosci drzewa
19 optimistic_value_estimation <- function(node) {
20   if (node$capacity > K) {
21     return(0)
22   }
23
24   estimate <- node$value
25   new_level <- node$level + 1
26
27   for (i in new_level:N) {
28     estimate <- estimate + item_values[i]
29   }
30
31   return(estimate)
32 }
33
34 DFS <- function() {
35   max_profit <- 0  # ostateczna wartosc plecaka
36   not_visited <- list()  # lista wezlow
37   branching_count <- 0  # liczba rozszerzonych wezlow
38   opt_items <- c()  # optymalne przedmioty w plecaku
39
40   # definicja wezla root (glowny pierwszy wezel)
41   root <- Node(0, 0, 0)
42   root$opt_estimate <- optimistic_value_estimation(root)
43   not_visited <- append(not_visited, list(root),1)
44
45   while (length(not_visited) > 0) {
46     # wyciaganie wezla z listy nieodwiedzonych wezlow
47     parent <- not_visited[[1]]
48     not_visited[[1]] <- NULL
49     cat(parent$level, parent$value, "\n")
50     branching_count <- branching_count + 1
51     # czy optymistyczna wartosc wezla moze dac wiekszy zisk niz obecny najlepszy wezel
52     if (parent$opt_estimate > max_profit) {
53       ### BRANCH LEFT, rozwazamy dodany przedmiotu
54       child <- Node(
55         parent$level + 1, # zwiekszenie poziomu wezlu
56         parent$value + item_values[parent$level + 1],  # zwiekszenie wartosci plecaka
57         parent$capacity + item_weights[parent$level + 1]  # zwiekszenie wagi plecaka
58       )
59       # dodanie przedmiotow rodzica do plecaka
60       child$items <- c(parent$items, parent$level + 1)
61       # wyznaczenie optymistycznej wartosci
62       child$opt_estimate <- optimistic_value_estimation(child)
63
64       # zapisanie nowego najlepszego wezla jesli jest poprawnym rozwiazaniem
65       if (child$capacity <= K && child$value > max_profit) {
66         max_profit <- child$value
67         opt_items <- child$items
68       }
69       # jezeli wciaz mozemy zyskac na podstawie optymistycznej wartosci to rozszerzeamy (
     depth first search, dodajemy na poczatek listy)
70       if (child$opt_estimate > max_profit && child$level < N) {
71         not_visited <- append(list(child), not_visited, 1)
72       }
73
74       ### BRANCH RIGHT, nie dodajemy przedmiotu do plecaka
75       child2 <- Node(
76         parent$level + 1,
77         parent$value,
78         parent$capacity
79       )
```

```
      child2$items <- parent$items
      child2$opt_estimate <- optimistic_value_estimation(child2)

      # zapisanie nowego najlepszego wezla jesli jest poprawnym rozwiazaniem
      if (child2$capacity <= K && child2$value > max_profit) {
        max_profit <- child2$value
        opt_items <- child2$items
      }
      # jezeli wciaz mozemy zyskac na podstawie optymistycznej wartosci to rozszerzeamy (depth first search, dodajemy ale nie na sam poczatek, aby wrocic po skonczeniu obecnego brancha)
      if (child2$opt_estimate > max_profit && child2$level < N) {
        not_visited <- append(list(child2), not_visited, 2)
      }
    }
  }

  return(list(max_profit = max_profit, opt_items = opt_items, branching_count = branching_count))
}

result <- DFS()
print(result)
```

Listing 31: Problem plecakowy algorytm branch and bound