131890-S Optimization methods **Winter 2024/2025**

Lecture 1: Introduction to analytical optimization

Daniel Kaszyński

Organisation

- **Instructor**: mgr Daniel Kaszyński, dkaszy[@]sgh.waw.pl
- **Consultation**: Monday 15:20-16:40, G-206.
- **Course grading**: A written exam on the content presented in class or included in the materials

  Example task: Describe method XYZ, show differences between ABC and XYZ, solve an analytical optimization task, carry out two iterations of the XYZ method
- **Required literature**:
  - [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.
  - [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 Definition of extremum

Extremum is the central concept concerning the analytical optimization problem. The extremum x^* is the solution of the **evaluation function** f for which x^* generates 'the best' solution. Let's consider a function of one variable mapping from real to real values $f : \mathbb{R} \to \mathbb{R}$.

Definition 1: Local extremum

By a local extremum (type minimum) we will call a point x^* for which:

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

Definition 2: Global extremum

By a global extremum (type minimum) we will call a point x^* for which:

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

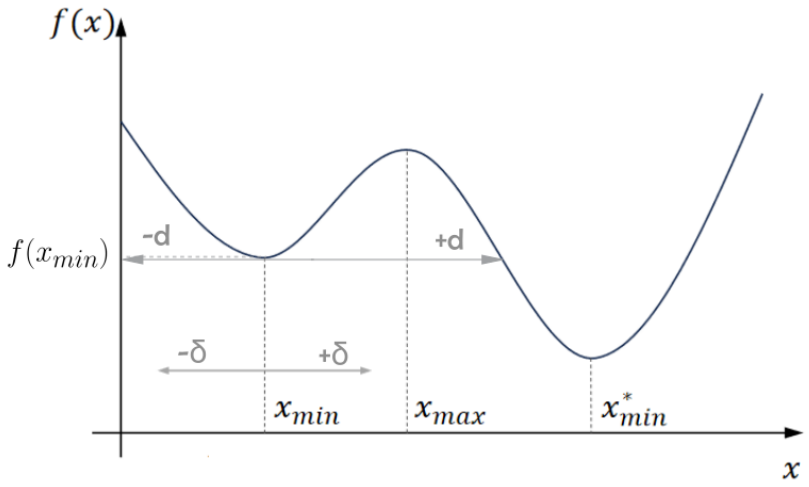


Figure 1.1: Local extremum of a function $f : \mathbb{R} \to \mathbb{R}$

Notice that the difference between a *local* and *global* extremum is the area in which we obtain better solution (from the point of view of the evaluation function). In case of a local extrema, we can point a neighborhood around x_{min} in which we obtain worse solutions. The neighborhood for x_{min} might be very small, where as for a global extremum x^*_{min} it's any neighborhood around an extremum.

1.2 Non-linear optimization without constraints in 1D. $f : \mathbb{R} \to \mathbb{R}$

The basic definition related to the non-linear optimization are related to the definition of derivative of a function.

Definition 3: Derivative of a function

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ we call a function:

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

Example 1. Let $f(x) = x^2 + 2x$. Calculate derivative of a function at the point $x_0 = 2$ from the definition:

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

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

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

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

Continuity of a function f is a necessary condition for a function to be differentiable!

In an analogous way we can describe the second derivative of a function:

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

1.2.1 First Order Conditions $f : \mathbb{R} \to \mathbb{R}$

Theorem 1: First Order Conditions $f : \mathbb{R} \to \mathbb{R}$**.**
Let $f : \mathbb{D} \subset \mathbb{R} \to \mathbb{R}$, $f \in C^1$. *If a function* f *has an extremum at the point* $x \in \mathbb{D}$, *then* $f'(x) = 0$.

Proof. If a function f has a minimum extremum at the point x, then there must exist $|d| > 0$, such that for all $0 < |\delta| < |d|$, we have $f(x+\delta) > f(x)$, or $f(x+\delta) - f(x) > 0$. Dividing both sides by a δ we obtain:

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

for $\delta > 0$ i $\delta < 0$. Respectively at the limit $\delta \to 0^+$ i $\delta \to 0^-$ we have:

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

If a function f is differentiable then $f'(x) = f'_+(x) = f'_-(x) = 0$. □

First Order Conditions give us a way to filter solutions from the search space, to those where the first derivative of a function is equal to zero (stationary points).

Caution! Just because a derivative of a function at the point x is equal to zero, doesn't mean that it is an extremum. For an example consider functions $f(x) = x^2$ and $f(x) = x^3$.

1.2.2 Taylor's Theorem $f : \mathbb{R} \to \mathbb{R}$

We will now introduce one of the most important theorem of mathematical analysis. The **Fundamental theorem of calculus** states following:

Theorem 2: Fundamental theorem of calculus.

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

where $F(x)$ is an anti-derivative or indefinite integral at point X. It means that the area under the curve of a function f between points a and b we have to calculate: (1) the area under the curve from $-\infty$ to b, (2) the area under the curve from $-\infty$ to a, (3) subtracting these values. Intuition behind equation (1.9) is shown on Figure 1.2.

To simplify symbols let us assume that $\int f'(x)\, dx = f(x)$, then we can rewrite (1.9) as:

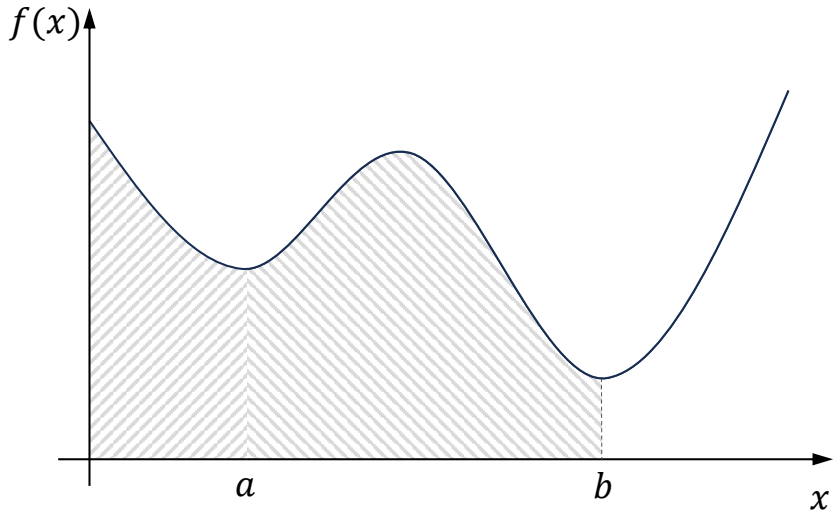


Figure 1.2: Definite integral

$$f(b) - f(a) = \int_a^b f'(x)\, dx \tag{1.10}$$

Moving forward we will use symbols $a = x$ $b = x + h$. We use point a as a starting point (or reference point), and point b as an offset by h from a. Notice that if we set $h = 0$ and start increasing it, then equation (1.10) answers the question by how much the area under the function f increases.

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

By rearranging this equation and bringing the indefinite integral to the beginning of a coordinate system (as for now it was attached at the point x) we obtain:

$$f(x + h) = f(x) + \int_0^h f'(x + a)\, da \tag{1.11}$$

The equation (1.11) is important from the perspective of further transformations. We can see that x is interpreted as a constant value (as the integral is on a). Also, the left side of the equation is in the similar form of a function inside the integral. Let's try to express the integral in terms of equation (1.11):

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

Using the addition property of an integral:

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

We can try to express the first integral in the following form:

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

Which gives us:

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

The expression inside the double integral we can also express using the previously noticed property:

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

The inside of the double integral can be written as:

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

Introducing this equation we obtain:

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

The equation inside the integral is always worked out of equation 1.11. Such procedure can be performed indefinitely (technically as many times as the function f is differentiable). In general this equation is given as:

$$f(x+h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}h^n \tag{1.12}$$

The equation (1.12) is called as a *Taylor's equation*. Notice that in practice we usually won't differentiate f infinitely many times. We will do it only few times that satisfies us with its accuracy. Thus we can write out N-th expansions of a function using Taylor's equation:

$$f(x+h) = f(x) + \sum_{n=1}^{N-1} \frac{1}{n!}f^{(n)}(x)h^n + \frac{f^{(N)}(x+\theta h)}{N!}h^N \tag{1.13}$$

where $R_N(x,h) = \frac{f^{(N)}(x+\theta h)}{N!}h^N$ is the Lagrange remainder. We can show that, this remainder has the following property:

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

Which means that the remainder $R_N(x,h)$ of approximation using Taylor's equation decreases to 0 at a rate faster than the polynomial of N-th order.

Theorem 3: Taylor's equation $f : \mathbb{R} \to \mathbb{R}$**.**
Let $f : \mathbb{D} \subset \mathbb{R} \to \mathbb{R}$ *and* $f \in C^N$ *at every point of the segment* $[x, x+h]$*. Then for some* θ *we have:*

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

Taylor's theorem is an important result that is often used in practice!

Expansion of a function using a Taylor's series of 1st and 2nd order:

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

Example 2. Expand function $f(x) = \sin x$ using Taylor's series around point $x_0 = 0$.

Compute the derivatives of a function:

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

For higher order derivatives this pattern continues, now evaluate derivatives at 0

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

Using Taylor's formula we have:

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

Using programming language **R** we can expand a function $f(x) = \frac{x^2}{e^x}$ into a Taylor's series of 1st and 2nd order:

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

# Numerical derivatives
d1f <- function(f, x, h = 10^-6) (f(x+h)-f(x))/h
```

```r
d2f <- function(f, x, h = 10^-6) (f(x+2*h)-2*f(x+h)+f(x))/h^2

# Taylor approximation of function f around 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

# Plots
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: Example: expanding function into a Taylor's series

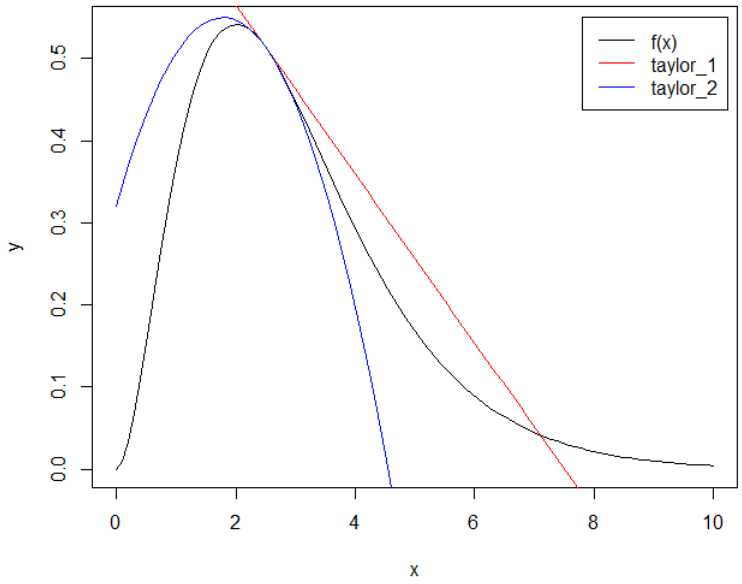


Figure 1.3: Example: expanding function into a Taylor's series

1.2.3 Second Order Conditions $f : \mathbb{R} \to \mathbb{R}$

We showed earlier that First Order Conditions are the required (each extremum has such property), but not enough (points that are not extremas, but have such property). To if a stationary point has an extremum we will use *sufficient conditions* – Second Order Conditions.

Theorem 4: Second Order Conditions $f : \mathbb{R} \to \mathbb{R}$**.**
Let $f : \mathbb{D} \subset \mathbb{R} \to \mathbb{R}$, $f \in C^n$. *If for some* $x \in \mathbb{D}$ *we get:* $f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$, *but for* $f^{(n)}(x) \neq 0$, *then:*

1. *If* n *is even, then function* f *has an extremum at the point* x*; If* $f^{(n)}(x) > 0$ *then it is a minimum, if* $f^{(n)}(x) < 0$ *then it is a maximum.*

2. If n is odd, then function f doesn't have an extremum at the point x.

Proof. From the Taylor's equation, for some $0 < \theta < 1$ we have:

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

because $f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$ then:

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

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

When n is even and $f^{(n)}(x) > 0$ then due to parity of n we have $h^n > 0$. Due to continuity of the function $f^{(n)}$ at the point x we know that for some $\delta > 0$ such, that for each $h : 0 < |h| < \delta$ we have $f^{(n)}(x+h) > 0$, due to that $f^{(n)}(x+\theta h) > 0$. What it means is that a function f has in this point x a minimum. We can show analogously the maximum case. □

1.3 Non-linear optimization without constraints, *multivariate* case, $f : \mathbb{R}^n \to \mathbb{R}$

In the previous section we showed the optimization conditions in a one dimensional case – one decision variable. However, the nature of optimization problems is more complicated and is multivariate.

We first have to introduce some basic terms from the area of differential calculus related to multiple variables. We recommended to get familiar with early chapters of a textbook [BI86].

Definition 4: Directional derivative

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Directional derivative of a function f at the point x in direction h we call the function:

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

Definition 5: Partial derivative

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Partial derivative of f at the point x with respect to variable x_i, $i = 1, 2, \ldots n$ we call the function:

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

where e_i is the i-th versor of space $\mathbb{R}^n$. Partial derivative of f with respect to x_i is then a directional derivative of f in direction of i-th versor, meaning that $h = e_i$.

Definition 6: Gradient of a function

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. By a gradient of a function f we call function $\nabla_f(x) : \mathbb{R}^n \to \mathbb{R}^n$ at the point f x:

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

Relation of directional derivative and gradient? If the gradient of a function $\nabla_f(\mathbf{x})$ exists at the point $\mathbf{x}$ (which means that the function f is differentiable in $\mathbf{x}$)

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

then directional derivative of a function f in direction of a vector $\mathbf{h}$ is equal to the dot product of a gradient $\nabla_f(\mathbf{x})$ and vector $\mathbf{h}$:

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

1.3.1 First Order Conditions $f : \mathbb{R}^n \to \mathbb{R}$

Theorem 5: First Order Conditions $f : \mathbb{R}^n \to \mathbb{R}$.
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$. If a function f has an extremum in point x, then $\nabla_f(x) = \mathbf{0}$

Proof. Let's consider a function $g_x(t) = f(x + th)$ and $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Because f has an extremum in x, then g has an extremum at $t = 0$, then $g'(t) = 0$, which means that $g'(t)|_{t=0} = 0$. As a result:

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

□

Example 3. Let's consider a function $f(x) = x_1^2 + x_2^2$. Find extremum of $f(x)$.

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

1.3.2 Second Order Conditions $f : \mathbb{R}^n \to \mathbb{R}$

To derive Second Order Conditions for a multivariate function we have to introduce a generalization of a second derivative of a function, namely Hessian matrix.

Definition 7: Hessian matrix

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. By a Hessian matrix $H_f(x)$ we call a matrix:

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

Caution! Hessian matrix, is a symmetric matrix only, when all second order partial derivatives are continuous (Schwarz's theorem). Which means that: $\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{\partial^2 f}{\partial x_j \partial x_i}(x)$

Also, the previously introduced Taylor's equation for a one dimensional case, can be redefined for the multivariate case:

Theorem 6: Taylor's theorem $f : \mathbb{R}^n \to \mathbb{R}$**.**
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$ and $f \in C^2$ in each point of a section $[x, x + h]$. Then:

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

Theorem 7: Second Order Conditions $f : \mathbb{R}^n \to \mathbb{R}$**.**
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^2$. If in x^* we have both:

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

Then w x^* is a local minimum of f.

Proof. From the Taylor's theorem for $\mathbb{R}^n$ we have:

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

where $f(x+h) - f(x) = \frac{1}{2}h^T H_f(x)h + o(|h|^2)$. From the Rayleigh's theorem, value of a quadratic form $h^T H_f(x)h$ can be bounded from below by a $\lambda_{min}|h|^2$:

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

For small enough h we get $f(x + h) - f(x) > 0$ □

What is left is to tell what happens when the matrix is positive definite $H_f(x^*) > 0$?

Theorem 8: Sylvester's criterion.
Let A be a symmetric real values matrix:

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

We can define first minors of a matrci A as:

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

Then:

1. *Matrix A is positively defined if and only if all first minors M_i of A are positive.*
2. *Matrix is negatively defined if and only if all even first minors M_i of A are positive, and all odd minors M_i are negative.*

Example 4. a Let $f(x) = x_1^2 + x_2^2$. Find extrema of $f(x)$:

First Order Conditions:

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

Second Order Conditions:

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

The $f(x)$ function has only one extremum (minimum) at point $[0, 0]$.

131890-S Optimization methods **Winter 2024/2025**

Lecture 2: Analytical optimization with constraints

Daniel Kaszyński

2.4 Properties of gradient

We have already introduced the concept of the gradient of a function – which is a vector of partial derivatives. Gradient will be quite often used as part of a lecture, which is why we should consider its properties. As a remainder:

Definition 8: Gradient

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$. Through a gradient of function f, we call function $\nabla_f(x) : \mathbb{R}^n \to \mathbb{R}^n$ at point x:

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

Remember that:

Relationship between Directional Derivative and Gradient? If the gradient of a function exists $\nabla_f(\mathbf{x})$ at point $\mathbf{x}$ (which means that f is differentiable in $\mathbf{x}$)

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

then directional derivative of a function f in direction of a vector $\mathbf{h}$ is equal to the dot product of the gradient ∇f and vector $\mathbf{h}$.

Theorem 9: Gradient is the direction of the fastest growth.

Proof. Let $|h| = 1$, i.e. let it be a normalized vector. Then the growth rate of a function f at point x in direction h is given by a directional derivative $\frac{df}{dh}(x)$. Let us determine then a direction h, which maximizes the growth rate of a function f, i.e. direction which maximizes the directional derivative:

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

for $|\nabla_f(x)| \geq 0$ and $cos(\nabla_f(x), h) \in [-1, 1]$ the growth rate of f is the greatest when $cos(\nabla_f(x), h) = 1$, which implies that h points in the same direction that $\nabla_f(x)$ is. As a result $h = \frac{\nabla_f(x)}{|\nabla_f(x)|}$. □

Theorem 10: Gradient is orthogonal to the level set of the function.

Proof. Let $f : \mathbb{R}^n \to \mathbb{R}$, $x^* = (x_1^*, \ldots, x_n^*)$ and $\nabla_f(x^*) \neq 0$. Let $r : \mathbb{R} \to \mathbb{R}^n$ so that $r(t_0) = x^*$. Value of the function is constant for all points from a chosen level set (according to the definition) and $\forall_{t\in\mathbb{R}} f(r(t)) = c$. Then $\frac{d}{dt}(f(r(t)) = \nabla_f(r(t))\frac{dr}{dt}(t) = 0$. In particular $\nabla_f(r(t_0))\frac{dr}{dt}(t_0) = 0$.

Because $\frac{dr}{dt}(t_0)$ is a tangent space to the level set of a function f at x^*, it implies that $\nabla_f(x^*)$ is orthogonal to the level set. □

2.4.1 First Order Conditions, equality constraints

Theorem 11: Lagrange theorem, Method of Lagrange Multipliers.
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$. If function f has an extremum at x related to $h(x) = 0$, where $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^1$, at point x, then

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

For $h : \mathbb{R}^n \to \mathbb{R}$ (when there is only one constraint).

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

Example 5. Let $f(x) = x_1^2 + x_2^2$, and $h(x) = x_1^2 + 2x_2^2 - 1$. Find the extremum of $f(x)$ related to $h(x) = 1$.

From First Order Conditions (FOC) we get:

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

Which gives us 4 solutions:

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

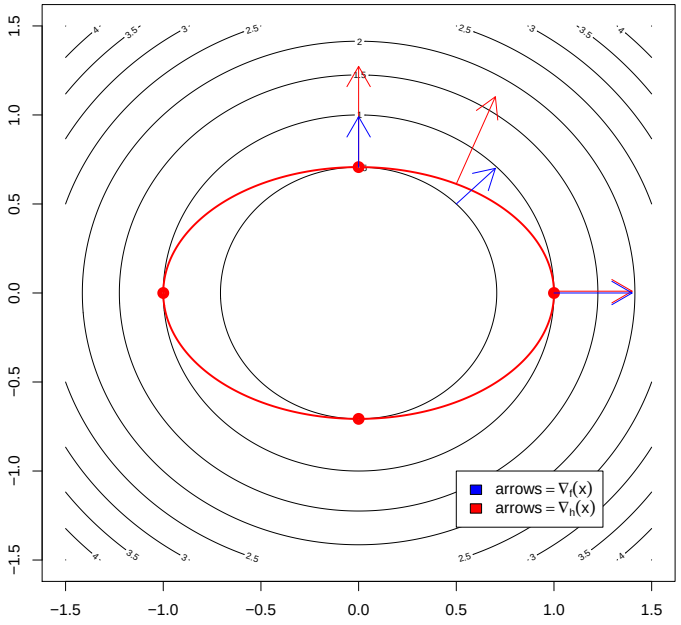


Figure 2.4: First Order Conditions equality constraints visualized

2.4.2 Second Order Conditions, equality constraints

Theorem 12: Second Order Conditions of Lagrange theorem.
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^2$. *Let there be* x *and* λ *such that:*

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

Then by point x *we call minimum of a function* f, *given that* $h(x) = 0$. $T(x)$ *we call tangent space, i.e.:* $T(x) = \{z \in \mathbb{R}^n : z^T \mathbf{D}h(x) = 0\}$. *In a case when* $m = 1$, *we have* $T(x) = \{z \in \mathbb{R}^n : z^T \nabla_h(x) = 0\}$

Example 6. Lets consider 4 solutions that we have found from 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 First Order Conditions, inequality constraints

Theorem 13: First order Karush-Kuhn-Tucker conditions, KKT theorem.
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $f \in C^1$ be an objective function, and let $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^1$ and $g : \mathbb{R}^n \to \mathbb{R}^p$, $g \in C^1$ be the constraints. If x is an extremum then there exists a pair of $\lambda = (\lambda_1,\ldots,\lambda_n)$ oraz $\mu = (\mu_1,\ldots,\mu_n)$ such that:

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* $\forall_{i=1,\ldots,p} g_i(x) \le 0$
3. ***Dual feasibility**:* $\forall_{i=1,\ldots,p}\ \mu_i \ge 0 \leftarrow$ *for minimum*
4. ***Complementary slackness**:* $\forall_{i=1,\ldots,p}\ \mu_i g_i(x) = 0$

Example 7. $f(x) = x_1^2 + x_2^2$ constrained by $[x_1,x_2] : g(x) = x_1^2 + 2x_2^2 - 1 \le 0$

From FOC we have:

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

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

Which gives 5 solutions:

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**

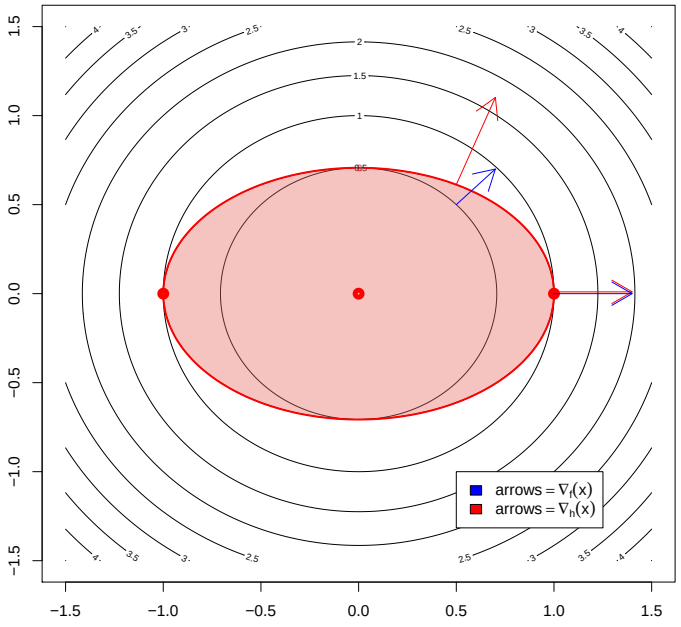


Figure 2.5: First Order Conditions inequality constraints visualized

2.4.4 Second Order Conditions, inequality constraints

Theorem 14: Second Order Conditions, KKT theorem.
Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $h : \mathbb{R}^n \to \mathbb{R}^m$, $h \in C^2$, $g : \mathbb{R}^n \to \mathbb{R}^p$, $g \in C^2$. *Let there be* x, λ *and* μ, *such that:*

1. $\nabla_f(x) + \lambda^T \mathbf{D}h(x) + \mu^T \mathbf{D}g(x) = 0$,

2. $\forall_{z \in T(x), z \neq 0}$ *we have* $z^T H_{L(x)} z > 0$

Then we call point x *a minimum of* f, *related to* $h(x) = 0$.

$T(x)$ *we call tangent space, i.e.:* $T(x) = \{z \in \mathbb{R}^n : z^T \mathbf{D}h(x) = 0\}$. *In a case where* $m = 1$, *we have* $T(x) = \{z \in \mathbb{R}^n : z^T \nabla_h(x) = 0\}$

131890-S Optimization methods **Winter 2024/2025**

Lecture 3: Simplex algorithm

Daniel Kaszyński

3.5 Linear programming

Linear programming is an area of optimization that enables solving the simplest optimization problems - linear problems. It is a method of obtaining the optimal result for linear models:

- Linear objective function,
- Linear equality or inequality constraints.

It's widely used in various fields such as operations research, economics, engineering, and management science to solve allocation, scheduling, and resource allocation problems efficiently.

The set of feasible solutions is a convex set (defined by a finite set of intersecting hallf-spaces - each determined by a linear inequality).

The objective function is an affine function of a real variable defined on this polyhedron. Linear programming allows finding a point on this polyhedron that gives the smallest (largest) value of the objective function. In linear programming, the objective function and the constraints are all linear. The objective function represents the quantity that needs to be optimized, while the constraints represent limitations or conditions that must be satisfied. The goal is to find the values of decision variables that optimize the objective function while satisfying all the constraints.

The general form of a linear programming problem is as follows:

- Find vector x
- that maximizes $c^T x$
- subject to $Ax \leq b$
- and $x \geq 0$

where the vector x which components are to be determined, c is given vector of values that we want to optimize, b is given vector of elements that we are constraining on and A is given matrix of variables next to constraints. The function $c^T x$ whose value is to be maximized is called the objective function. The convex polytope, over which the objective function is to be optimized, is constructed using constraints $Ax \leq b$ and $x \geq 0$.

3.6 Simplex algorithm

3.6.1 Algorithm basics

The **simplex algorithm** is a widely used method for solving linear programming problems. It's often considered one of the most efficient algorithms for solving linear problems in practice.

The simplex algorithm works by iteratively moving from one vertex (corner point) of the feasible region to another along the edges of the polytope defined by the constraints until it reaches the optimal solution. At each step, the algorithm selects a pivot element to improve the objective function value and moves to the adjacent vertex that corresponds to the pivot element.

The simplex algorithm uses standard form to represent inequalities. Inequalities are thus converted to equalities containing an additional **slack variables**.

Example 8. Let $f(x, y, z) = -x + 3y + 2z$ be a function that we want to maximize under following conditions:

- $x + y + z \leq 6$
- $x + z \leq 4$
- $y + z \leq 3$
- $x + y \leq 2$

provided that $x, y, z \geq 0$. Inequality constrains represented in standard form using slack variables would look like this:

- $x + y + z + r + s + t + u = 6$
- $x + z + r + s + t + u = 4$
- $y + z + r + s + t + u = 3$
- $x + y + r + s + t + u = 2$

and function f would be equal to $-x + 3y + 2z + r + s + t + u$.

3.6.2 Simplex table

The simplex algorithm often uses the representation of the analyzed problem in the form of an **simplex table**.

Assuming that a function f that we want to maximize is equal to $-x + 3y + 2z$ and we maximize under following conditions:

- $x + y + z \leq 6$
- $x + z \leq 4$
- $y + z \leq 3$
- $x + y \leq 2$
- $x, y, z \geq 0$

example simplex table for this case would look like this:

Table 3.1: Example simplex table

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

In the created table, the area marked in yellow is matrix A - matrix of factors next to inequality constraints. In the bottom row we see factors of the objective function. The column on the right side of the table contains the inequality constrain values - b (values on the right side of inequalities). Last but not least, we also have 4 slack variables, one for each constrain. Identity matrix for those variables is marked in pink in the created table.

If the column in the table corresponding to a given variable contains all zeros and a single one, the given variable will be non-zero. Otherwise, this variable will be zero.

Initial simplex table is constructed in a way that the variables x, y, z are equal to zero (also called non-basic) and slack variables are non-zero (also called basic).

3.6.3 Gaussian elimination algorithm

Next step in the simplex algorithm is usage of the Gaussian elimination algorithm.

The **Gaussian elimination algorithm** is a method used to solve systems of linear equations by transforming the augmented matrix representing the system into row-echelon form through a sequence of elementary row operations. It is a fundamental technique in linear algebra and is used in various applications such as solving systems of linear equations, computing matrix inverses, and finding eigenvalues and eigenvectors. Being efficient and numerically stable, it is widely used method for solving linear algebraic problems.

As the first step of Gaussian elimination we need to select the column (or variable) with the biggest value in the bottom row of the table. If several variables are in a tie for the largest value, we can choose any of them. The selected variable will be the new basic variable entering the basis and it's corresponding column can be referred to by *key column* or *pivot column*. Continuing with the example from subsection about simplex table, the variable y has the greatest impact (since it is multiplied by 3).

Next we need to determine which slack variable will be replaced by the selected variable. To do that we should calculate ratio of b column (the right-most column) to selected variable. Fortunately, all the values of the y column are either 1 or 0 in our case. We should choose the row with **minimum ratio**, so in this case the last constraint. The row with the minimum ratio value is called *key row* or *pivot row*.

Table 3.2: Pivot row and column in example simplex table

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

The following step is to perform a series of elementary row operations so that the pivot column becomes

a unit column with 1 in the intersection with pivot row (*pivot element*). To achieve this, we must first divide all elements of the pivot row by the pivot element (in our example pivot element is equal to 1 so the pivot row stay the same). Then we divide the remaining rows by the pivot row to get zeros in the pivot column.

Table 3.3: Example simplex table after first iteration of the algorithm

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

These steps are repeated until all values in the bottom row are equal to or less than zero. This ensures that the value of the analyzed solution cannot be improved any further.

Table 3.4: Example simplex table after all iterations of the algorithm

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

The final solution for this example is $x = 0$, $y = 2$ and $z = 1$. The variable x is zero because it is not a unit column while other variables' values can be read from the right-most column.

3.6.4 Simplex algorithm implementation

To implement the simplex algorithm, we must first prepare a function responsible for creating a simplex table. It will first read the number of given variables and then prepare a matrix of values for the initial simplex table. Using the **R** programming language, the example implementation of this function could look like this:

```
create_simplex_table <- function(A_mat, b_vec, c_vec){
  # Read number of variables/constraints
  n_var <- length(c_vec)
  n_con <- length(b_vec)

  # Construct Simplex Table
  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 2: Implementation of the function responsible for creating a simplex table

The simplex algorithm also requires Gaussian elimination algorithm to be implemented. A single step of this method will be responsible for selecting a pivot column and a pivot row and using them to transform selected column into unit column. It is worth noting that for this purpose we can use the built-in **R** method - *outer()*. It enables you to create a new matrix or array by applying a function to every conceivable combination of the items from two input vectors (where default function is multiplication). Implementation of Gaussian elimination algorithm could be created as follows:

```
gaussian_elimination <- function(st, b_vec, c_vec){
  # Read number of variables/constraints
  n_var <- length(c_vec)
  n_con <- length(b_vec)

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

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

  # Gaussian Elimination
  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 3: Implementation of the Gaussian elimination algorithm

Of course, at the end of the algorithm we would also like to be able to read the results of our calculations. A method to read the results based on the simplex table obtained may be helpful. For example, such a method might look like this:

```
read_results <- function(st, b_vec, c_vec){
  # Read results
  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 4: Implementation of the function responsible for reading results

Having all the above implementations of the necessary functions, we can start implementing the algorithm itself. The simplex algorithm should contain the following steps:

- creating the initial simplex table,

- repeating the Gaussian elimination algorithm iterations until all values in the bottom row are equal to or less than zero,
- alternatively, the Gaussian elimination algorithm should end after a predetermined maximum number of steps k,
- reading the results.

The final implementation of the algorithm looks as follows:

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

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

  # Read outputs
  out <- read_results(st, b_vec, c_vec)
  return(out)
}
```

Listing 5: Implementation of the Simplex algorithm

131890-S Optimization methods **Winter 2024/2025**

Lecture 4: Numerical approximations

Daniel Kaszyński

4.7 Finite differences

A finite difference is a mathematical expression of the form $f(x+b) - f(x+a)$. If we divide the finite difference by $b-a$, we get the difference quotient. They are often used in finite difference methods for the numerical solution of differential equations.

4.7.1 Forward and backward differences

The most popular derivative approximation methods are forward and backward finite differences.

Definition 9: Derivative of a function

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **forward finite difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4.14}$$

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **backward finite difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{4.15}$$

Numerical differentiation algorithms estimating the derivative of a mathematical function using forward finite difference or backward finite difference have error $O(h)$ (this can be proven using Taylor's Theorem).

From Taylor's Theorem we know that:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{4.16}$$

We are able to transform the expression into:

$$f'(x)h = f(x+h) - f(x) - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 - ... \tag{4.17}$$

and dividing this expression by h we get:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{4.18}$$

So we can deduce that the error in the forward difference is of order $O(h)$.

Using the same methodology, we can transform the formula below:

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{4.19}$$

into a formula for the backward difference:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{4.20}$$

The backward difference has also error of order $O(h)$.

4.7.2 Central difference

In addition to these two methods of approximating the derivative of a function, we can also show a third one - using **central difference**. This method is sometimes called a symmetric.

Definition 10: Central derivative of a function

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **central difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{4.21}$$

Numerical differentiation algorithms estimating the derivative of a mathematical function using central difference have error $O(h^2)$. This is preferable as the error is smaller than previous methods (remember that we assume $h \to 0$).

Similarly to the previously described approximations, we can use Taylor's Theorem to derive a formula for central difference. However, this time we will need two starting equations:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{4.22}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{4.23}$$

By subtracting the two formulas above, we get:

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{1}{3}f'''(x)h^3 + ... \tag{4.24}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}f'''(x)h^3 - ... \tag{4.25}$$

4.8 Subtractive cancellation error

When working on calculating derivatives of functions, we may encounter not only mathematical but also technical/hardware problems. One popular problem of this nature is **subtractive cancellation error**.

Subtractive cancellation error can occur while dealing with floating-point arithmetic. When computers work with real numbers, they must somehow store a potentially infinite number of decimal places in those numbers. For this purpose, they use, among others, float variables, which are a finite approximation of real numbers. Most programming languages use a technical standard for floating-point arithmetic called **IEEE 754**. Unfortunately, precision is partially lost this way, but this is due to the computer's finite memory.

Subtractive cancellation error is a phenomenon that may be present when subtracting two nearly equal numbers. Floating-point numbers in computers have limited precision, and when you subtract two numbers that are very close in value, the result may suffer from loss of significant digits.

Example 9. Let $a = 0.3 + 0.3 + 0.4 - 1$ and $b = -1 + 0.3 + 0.3 + 0.4$.

Following simple mathematics, we can conclude that a is equal to b. Unfortunately, calculations performed on float variables may not give us the same result.

```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 6: Subtraction cancellation error example in **R** language

In this example, calculations performed to get a and b may result in very close, but different values. Those results might not be as accurate as one might expect due to subtractive cancellation error. The precision of the result depends on the number of significant digits that can be represented in the floating-point format. Therefore, we cannot assume that when subtracting float variables there will be no error which, although small, may spoil some simple comparisons and calculations.

To mitigate subtractive cancellation errors, various numerical analysis techniques and algorithms can be employed, such as rearranging the expression to avoid subtracting nearly equal numbers or using higher precision arithmetic when necessary. Additionally, understanding the limitations of floating-point arithmetic and being aware of potential sources of error is crucial when working with numerical computations in computer programs.

4.9 Complex Step Derivative

To avoid the problem of cancellation discussed in the previous section, various types of methodologies and formula transformations can be applied. One possible solution is to use **Complex Step Derivative**. The main idea behind this method is to take advantage of Taylor's equation and imaginary numbers to remove the need to subtract two float variables.

Let us start with Taylor's equation and let's use imaginary numbers to state it:

$$f(x + ih) = f(x) + f'(x)ih - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)ih^3 + ... \tag{4.26}$$

Assuming that we want to calculate the first derivative of the function, we need to transform the formula:

$$f'(x)ih = f(x + ih) - f(x) + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)ih^3 + ... \tag{4.27}$$

Then we need to isolate the derivative. We can start by dividing both sides by h:

$$f'(x)i = \frac{f(x + ih) - f(x)}{h} + \frac{1}{2}f''(x)h + \frac{1}{6}f'''(x)ih^2 + ... \tag{4.28}$$

To obtain only the derivative, we must also take care of the imaginary part:

$$f'(x) = Im\left(\frac{f(x + ih) - f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{4.29}$$

Fortunately, we were able to drop $\frac{1}{2}f''(x)h$ because it did not contain an imaginary part. This simplified our formula, but it's still not as good as we'd hope because we haven't gotten rid of the subtraction of two function instances. To do this, we should split the first part of our formula:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) - Im\left(\frac{f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{4.30}$$

We can notice that $Im\left(\frac{f(x)}{h}\right)$ has no imaginary part. So we can remove it from our equation. This leaves us with a formula that helps to avoid the subtractive cancellation error problem:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{4.31}$$

This formula is the essence of Complex Step Derivative. This is the fourth way we mentioned to calculate the derivative of a function. This method as an error of $O(h^2)$ and does not require the subtraction of two instance of the function f.

4.10 Comparison of the finite differences

To better understand the differences between the presented finite differences, it is worth conducting a series of tests for a simple function.

Let $f = sin(x^2)$. Using the rules of symbolic differentiation, we can work out that $f'(x) = 2xcos(x^2)$.

Let $u = x^2$. Then, $\frac{du}{dx} = 2x$ and $\frac{df}{du} = cos(u) = cos(x^2)$. If we put this information together, we get the following equation:

$$f'(x) = \frac{df}{dx} = \frac{du}{dx}\frac{df}{du} = 2xcos(x^2) \tag{4.32}$$

Just to be sure, we can use the **R** programming language to calculate the derivative of the function f. To do this, we must first import the `Deriv` library, which is used to calculate derivatives:

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

Listing 7: Import of `Deriv` library

When using external libraries, it is often worth checking their documentation. In **R** programming language it can be done by adding '?' sign in front of library name:

```
# Access to library documentation
?Deriv
```

Listing 8: Access to `Deriv` library documentation

Then, using this library, we can calculate the derivative of the function f:

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

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

Listing 9: Derivative of function f calculated using `Deriv` library

As a result of these calculations we get:

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

The plots of the function f and its derivative df would look as follows:

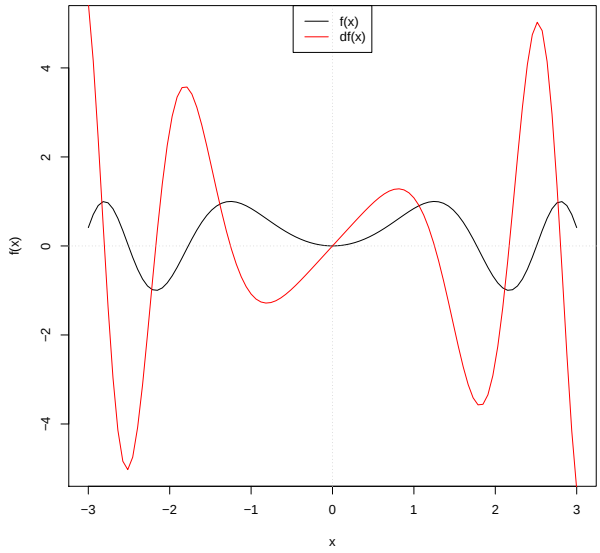


Figure 4.6: The plots of the function f and its derivative df

Of course, in order to compare different types of approximations of derivative functions in **R** programming language, we first need to have implementations of these methods. Knowing the definitions and formulas for these methods, creating these implementations is not a difficult task. For example, they may look like this:

```
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 10: Finite differences implementations

Next, we can check the differences in the values obtained by these approximations at the given point $x_0 = 1$. The following code snippet gives us values for each of finite differences:

```
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 11: Values obtained by finite differences in point $x_0 = 1$

```
df = 1.08060461173627953002
```

```
----------------------------------------

diff\_forward = 1.08060346903915416306

diff\_backward = 1.08060575443325035394

diff\_central = 1.0806046117917171340973

diff\_complex = 1.0806046117386829408 2
```

As we can see, these values are close to each other and do not deviate too much from the real value of the function's derivative. Method `diff_complex` achieves the best result because it is closest to the actual value of the derivative.

In addition to checking individual function values for each of the finite differences, we can do many other comparisons. For example, we can plot how fast relative error converges to 0 in terms of all of the numerical differences. The plot that would be created for those purposes should have exponential scale, because everything that is interesting to us will take place in a very narrow ranges. To improve visibility, logarithmic scale is also recommended.

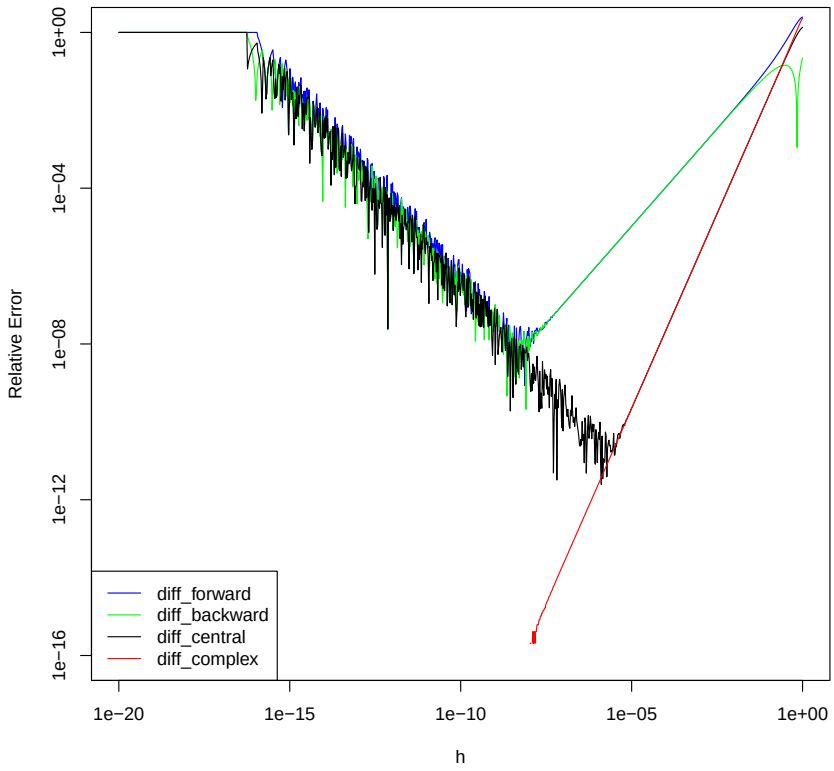


Figure 4.7: Relative error for each of the finite differences on f

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

```
3
4  # Plotting of the relative error for forward difference
5  plot(h, abs((df(x0) - diff_forward(f, x0, h)) / df(x0)), col = 'blue',
6      log = 'xy', type = 'l', ylab = 'Relative Error', ylim = c(10 ^- 16, 1))
7
8  # Adding to plot backward, central and complex differences
9  lines(h, abs((df(x0) - diff_backward(f, x0, h)) / df(x0)), col = 'green')
10 lines(h, abs((df(x0) - diff_central(f, x0, h)) / df(x0)), col = 'black')
11 lines(h, abs((df(x0) - diff_complex(f, x0, h)) / df(x0)), col = 'red')
12
13 # Adding legend to the plot
14 legend('bottomleft', c("diff_forward", "diff_backward", "diff_central", "diff_complex"),
15     col = c("blue" , "green" , "black" , "red"), lty = 1)
```

Listing 12: Relative error plot implementation

As we can see in the plot above, the characteristics of how relative error changes depending on h differ for each approximation method. By tracking the values relative to the h axis from right to left, we can see how relative error converges to zero (or in most cases just attempts to converge). For each finite difference we can define the following behavior:

- **diff_forward** and **diff_backward** - Forward finite difference and backward finite difference behaved very similar to each other while trying to converge relative error to zero. Both methods achieved a minimum error at an h of approximately $1e-08$. From that point on, however, the error began to increase. It was caused by described earlier subtractive cancellation error (look section 4.8). The error caused by subtracting two float variables for smaller h values significantly affected the quality of the results in those cases.
- **diff_central** - Central difference behaved similarly to the two previously described methods, with a few notable differences. First of all, at the beginning this method reached its minimum much faster at around $1e-05$. The slope of the relative error convergence for this approximation was much steeper. This was caused by the fact that central difference has error $O(h^2)$. Furthermore, the relative error values remained on average slightly below those obtained by forward and backward finite differences.
- **diff_complex** - Complex step derivative approximation performed the best out of all described methodologies. Not only did it converge to zero at a rate comparable to central difference, but also it didn't suffer from the problem of the subtractive cancellation error. This allowed this method to achieve relative error values close to zero to the point that the **R** language did not distinguish them from zero (which is why they disappeared from the plot).

4.11 Automatic differentiation

Automatic Differentiation, also known as algorithmic differentiation or autodiff, is a technique used to efficiently and accurately evaluate the derivatives of mathematical functions. The primary goal of this technique is to automatically and systematically compute the derivatives of a given function, making it especially useful in optimization, machine learning, and scientific computing.

Automatic differentiation sets itself apart from symbolic differentiation and numerical differentiation. Symbolic differentiation encounters challenges in converting a computer program into a unified mathematical expression, often resulting in inefficient code. On the other hand, numerical differentiation, employing the method of finite differences, may introduce round-off errors during the discretization process and face issues related to cancellation. These traditional methods struggle when calculating higher derivatives. In contrast, automatic differentiation effectively addresses and resolves all these issues.

To fully understand how automatic differentiation works, we must first become familiar with a few basic ideas behind it. First of all, the decomposition of differentials provided by the **chain rule** of partial derivatives is fundamental to automatic differentiation.

The chain rule is a concept in calculus that describes how to find the derivative of a composite function. Mathematically, if you have the composition of two functions $f(x)$ and $g(x)$ such that $f(g(x))$, then the chain rule states that the derivative of this composition with respect to x is the product of the derivative of f with respect to its argument $g(x)$ and the derivative of g with respect to x:

$$\frac{df}{dx}f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx} = f'(g(x))g'(x) \tag{4.33}$$

Another thing worth paying attention to is the fact that automatic differentiation uses computational graphs (explicitly or implicitly). A computational graph is a representation of a mathematical expression or a computational process. It is commonly used to visualize and understand the flow of computations involved in evaluating a function or performing a series of operations.

Most mathematical formulas can be broken down into a series of basic arithmetic operations (e.g., addition, multiplication, exponentiation). To create a computational graph, we first create nodes. Nodes in a computational graph represent mathematical operations or functions. Each node corresponds to a specific computation, such as addition, multiplication, or a more complex operation. Edges in the graph depict the flow of data or dependencies between the operations. An edge from one node to another indicates that the output of the first operation is used as an input for the second operation. The inputs to the computational graph are usually represented as nodes with no incoming edges, while the outputs are nodes with no outgoing edges.

Example 10. Let $f(x_1, x_2) = (\cos(\frac{x_1}{x_2}) + \frac{x_1}{x_2} - \sin(x_2))(\frac{x_1}{x_2} - \sin(x_2))$. Computational graph for such a function would look as follows:

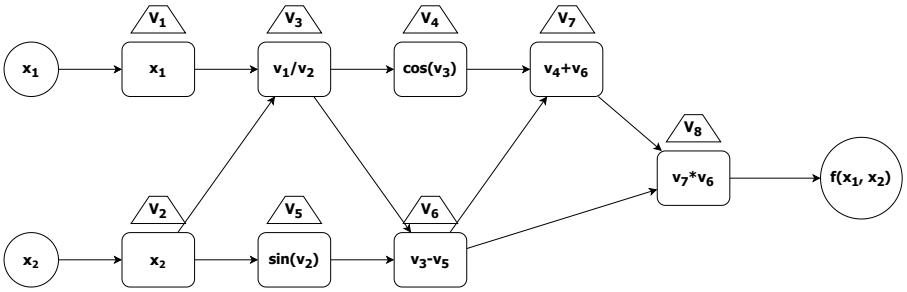


Figure 4.8: Example computational graph of function $f(x_1, x_2)$

Another important part of the algorithm is the use of *Dual Numbers*. In each node of the computational graph not only we will calculate primals of the function, but we will simultaneously compute their derivatives. This simple trick will help us reuse already calculated in previous steps components of the formula in future operations. At the same time, we limit ourselves to calculating derivatives of only simple arithmetic operations.

In the **R** programming language, we can represent such *Dual Numbers* using object-oriented programming:

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

```
    return(obj)
}
```

Listing 13: Dual Number implementation

Last but not least, there is also the issue of calculating the results of individual operations in the computational graph. An elegant way to approach this problem is to use **operator overloading**. For each operator (such as '+' or '-'), we can use it to define a different behavior specifically tailored to our *Dual Numbers*.

Lets create overloads for each of the basic operations. Firstly, we can start with addition operator '+':

```
"+" <- 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 14: Addition operator overload

Each operator is a function that takes two parameters as input and gives us the result. First, we should distinguish between the effect of the operator for *Dual Numbers* and other values (for which the effect of the operator will remain unchanged). If we want to add two *Dual Numbers*, we must both add their values and add the values of their derivatives. After that we can return newly created result as a *Dual Number*.

Proceeding in a similar way, we can implement operator overloading for the subtraction operator '-':

```
"-" <- 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 15: Subtraction operator overload

When overloading the multiplication operator - '*', we should recall *Leibniz product rule* (a formula used to find the derivatives of products of two functions):

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x) \tag{4.34}$$

```
"*" <- 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 16: Multiplication operator overload

As the last operator, that we will do for the sake of this example implementation, we can overload power operator. The formula for the derivative of power function may be helpful here:

$$f'(x) = \frac{d}{dx}(x^k) = kx^{k-1} \tag{4.35}$$

We just need to remember to multiply the value obtained using this formula by the previously calculated value of the derivative:

```
"^" <- 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 17: Power operator overload

Dual Numbers and overloaded operators are enough for the automatic differentiation algorithm to work. Now we can move on to testing the algorithm.

Example 11. Let $A = 5$ with sensitivity over 1-st variable, $B = 4$ with sensitivity over 2-st variable and $C = 7$ with sensitivity over 3-st variable. Furthermore, let $f(x_1, x_2, x_3) = x_1^2 + x_1x_2 + x_2 + x_3$:

```
# Declaration of DualNumer values
A <- DualNumber(5, c(1,0,0))
B <- DualNumber(4, c(0,1,0))
C <- DualNumber(7, c(0,0,1))

# Calculation of function f(A, B, C) values
A^2 + A*B + A + C
```

Listing 18: Automatic differentiation example

When we run this calculations, we would get following results:

$val

[1] 57

$eps

[1] 15 5 1

attr(,"class")

[1] "DualNumber"

When performing a simple operation on numbers of the *Dual Number* type, we can obtain both the value of the f function and the value of the derivative over each of the input variables. What's more, obtained values for *$eps* is precisely a gradient of function f in given point.

Lecture 5: Local methods

Daniel Kaszyński

5.12 Directional derivative and partial derivative

The directional derivative of a function measures how the function changes at a specific point in the direction of a given vector. In other words, it quantifies the rate of change of a function along a particular direction. It measures impact of the shift by vector h of function f.

Definition 11: Derivative of a 1D function

By a derivative of a 1D function $f : \mathbb{R} \to \mathbb{R}$ we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{5.36}$$

When dealing with a one-dimensional function, we implicitly assume that the vector h is simply equal to 1. This means that the shift in x domain is multiplied by 1 when moving in this space. In a more general case, especially when operating on multidimensional functions, we can move along different h vectors. They should be therefore included in the formula.

Definition 12: Directional derivative of a function

By a directional derivative of a multidimensional function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x+h \in \mathbb{D}$ at point x along vector h we call a function:

$$\frac{df}{dh}(x,h) = \lim_{\delta \to 0} \frac{f(x+\delta h) - f(x)}{\delta} \tag{5.37}$$

$$\frac{df}{dh}(x,h) = \nabla_f(x)h = |\nabla_f(x)||h| \cos(\alpha) \tag{5.38}$$

where $\nabla_f(x)$ is a gradient of a function f and α is an angle between vectors $\nabla_f(x)$ and h.

Partial derivative is a special case of directional derivative. A partial derivative describes the rate at which a multivariable function changes with respect to one of its variables while keeping the other variables constant. It is essentially the derivative of a function with respect to one of its independent variables, treating the other variables as constants. Partial derivative is also a sensitivity of a function.

Definition 13: Partial derivative

Let $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{D}$ and $h \in \mathbb{R}^n : x + h \in \mathbb{D}$. Partial derivative of f at the point x with respect to variable x_i, $i = 1, 2, \ldots n$ we call the function:

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

where e_i is the i-th versor of space $\mathbb{R}^n$. Partial derivative of f with respect to x_i is then a directional derivative of f in direction of i-th versor, meaning that $h = e_i$.

5.13 Kernel

Kernel (also known as the null space or nullspace) represents the set of solutions or vectors that "vanish" or map to the zero vector under the given linear transformation or matrix operation. The concept of the kernel is fundamental in linear algebra and is used in various applications, including solving systems of linear equations and understanding properties of linear transformations.

Consider a linear map represented as a m × n matrix A with coefficients in a field K, that is operating on column vectors x with n components over 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.

$$\mathrm{N}(A) = \mathrm{Null}(A) = \ker(A) = \{\mathbf{x} \in K^n \mid A\mathbf{x} = \mathbf{0}\}. \tag{5.39}$$

Kernel of n-dimentional space is an identity matrix of size n. In simpler words, it is the $n \times n$ square matrix with ones on the main diagonal (from top left to bottom right) and zeros elsewhere.

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

Each row of this matrix (also called versor of space) is named e_i, where $i = 1, 2, \ldots n$. They are useful while calculating partial derivative of a function (look Definition 4).

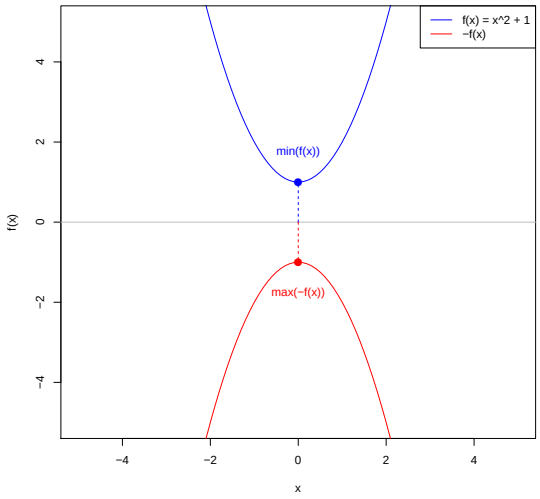
5.14 Optimality in multidimensional space

When working with multidimensional functions, we are often tasked with finding the extremum of those functions. If we are using local methods to do that, we should find direction or a vector that will point us towards said extremum. Local methods, as the name suggests, are considering only nearest neighbourhood of a point in space. Having that knowledge, the simplest way to find extremum is to follow direction of a quickest decrease (or increase) of a function. Vector that points in that direction is an optimal vector that we are searching for.

Optimality requires from us stating two conditions beforehand:

- Naming the scoring function
- Choosing if we want to minimize or maximize said function.

Scoring function in most cases is just a provided function f. We can also notice, that minimizing the function and maximizing the function are both very similar tasks. In fact, we can substitue maximizing the function with minimizing the inverse of this function.



$$\underset{x}{\operatorname{argmin}} f(x) = \underset{x}{\operatorname{argmax}}(-f(x)) \tag{5.40}$$

Example 12. Let $f(x, y) = x^2 + 2y^2$ and $x_0 = (2, 3)$. Plot for such a function would look as follows:

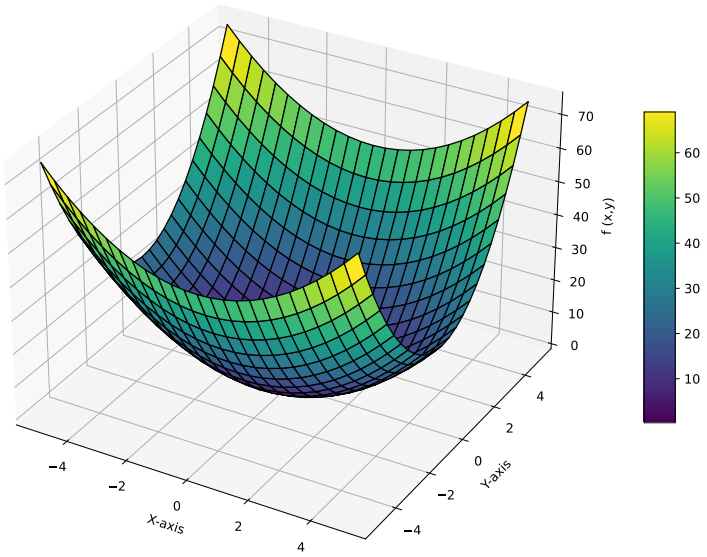


Figure 5.9: Function $f(x) = x^2 + 2y^2$

Function f has an extremum in point (0, 0). If we were to find that extremum from any starting point x_0 we would need to find an optimal vector that represents quickest decrease in function f. This vector is represented by the antigradient of this function $(-\nabla_f(x))$.

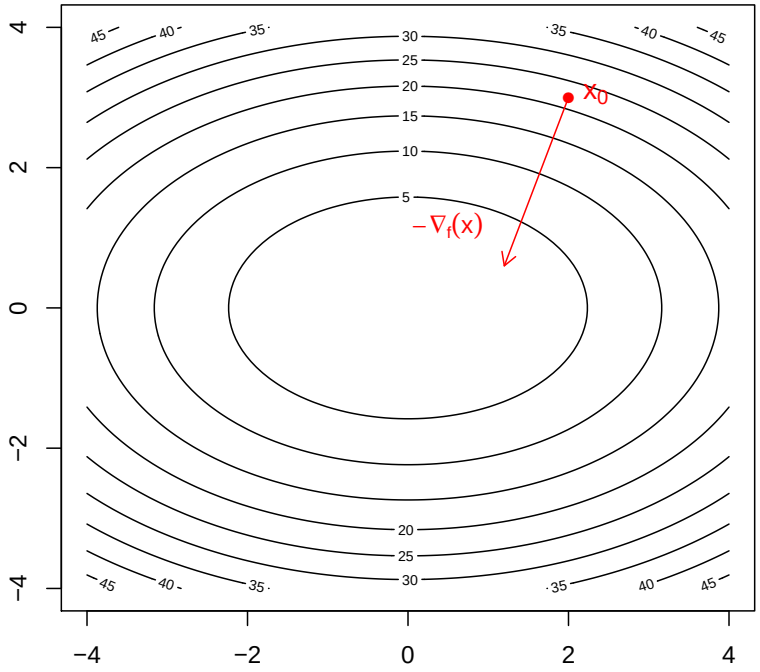


Figure 5.10: Vector starting at point x_0 that represents quickest decrease in function f

The directional derivative can help with finding the optimal vector. If we want to minimize a function (like in Example 1), lets consider mathematical formula for the directional derivative:

$$\underset{h}{\operatorname{argmin}} \frac{df}{dh}(x,h) = \underset{h}{\operatorname{argmin}} |\nabla_f(x)||h| \cos(\sphericalangle(\nabla_f(x),h)) \tag{5.41}$$

Assuming that the length of the gradient $\nabla_f(x)$ is constant and the length of the vector h is constant, the main parameter that we can use is the angle between these vectors $(\cos(\sphericalangle(\nabla_f(x),h)))$. The gradient shows us the direction of the fastest increase in the function value. Naturally, if we want to minimize the function, we should choose the opposite direction - antigradient. This is also confirmed by the formula above as $\cos(180^o)$ is equal to -1 while vector lengths are always positive values.

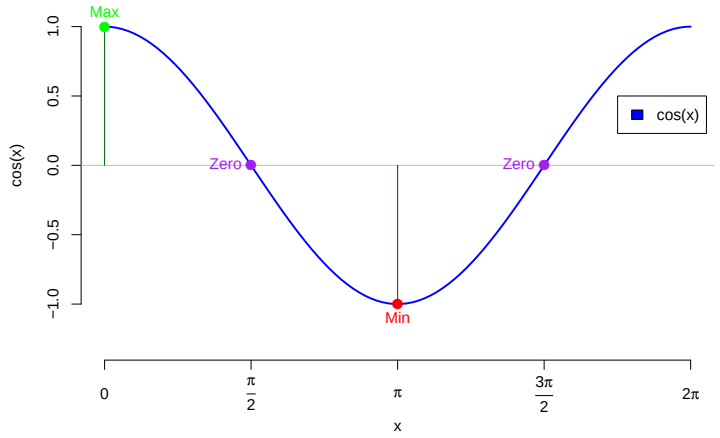


Figure 5.11: Cosinus function

5.15 Gradient descent

Gradient descent is an optimization algorithm commonly used to minimize nonlinear analytical functions. Those functions are often the cost or loss function in machine learning and other optimization problems. The goal of gradient descent is to iteratively move towards the minimum of a function by adjusting its parameters.

Definition 14: Gradient descent

Given that gradient descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - \alpha \nabla_f(x_k) \tag{5.42}$$

where $k \in \mathbb{N}$ is an iteration number, α is predefined learning rate and $\nabla_f(x_k)$ is the gradient of a function f in point x_k.

The general idea is to take repeated steps in the opposite direction of the gradient (or approximate numerical gradient) of the function at the each consecutive point, because this is the direction of steepest descent. On the other hand, stepping in the direction of the gradient will lead to a local maximum of that function.

Example 13. Knowing the formula for each step of gradient descent of function f, let $x_1 = x_0 - \alpha \nabla_f(x_0)$, where x_0 is a starting point, x_1 is the next point calculated in direction of biggest function decrease and parameter α is learning rate set to 0.1.

Following the same process, we can calculate the points even further down the line in the iterative manner:

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

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

Let $f(x) = x^2$ and $x_0 = 1$. Values for the next points x_0, x_1, x_2... in gradient descent iterations are equal to:

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

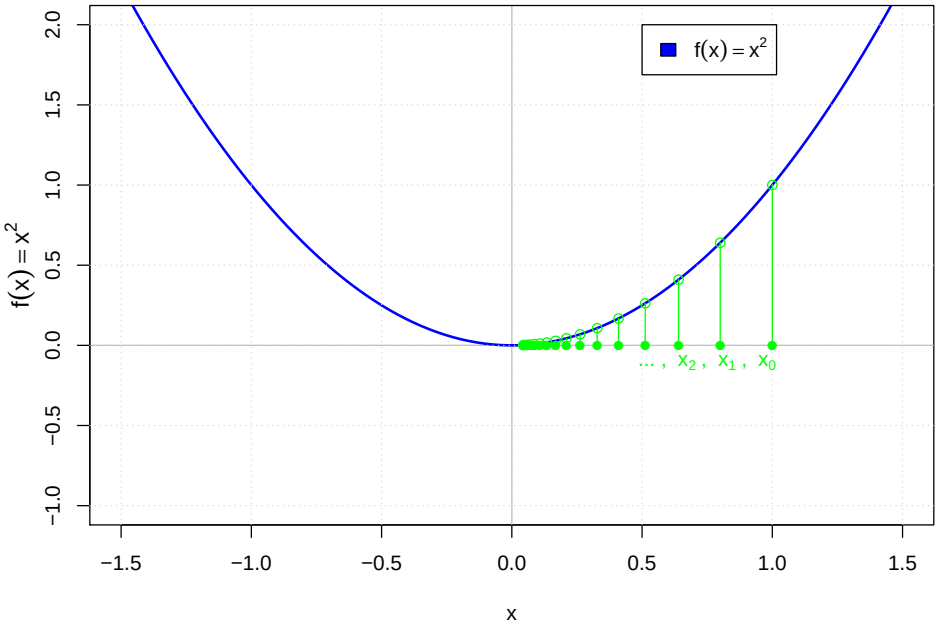


Figure 5.12: Converging of consecutive values of points x_0, x_1, x_2... to extremum of function $f(x) = x^2$ upon introducing learning parameter $\alpha = 0.1$

Using programming language **R** we can write a simple gradient descent implementation. First of all, we need a function to calculate the gradient. We could use $grad(f, x)$ from *numDeriv* library or we could implement it from scratch like so:

```
if(!require(docstring)) library(docstring) # A library for code documentation!

num_grad <- function(f, x, h = 10^-6){
  #' Central Numerical Gradient
  #'
  #' @description Function responsible for determining the numerical gradient
  #' by calculating the vector of central partial derivatives.
  #'
  #' @param f function. The function which gradient we determine.
  #' @param x numerical vector. The point at which the numerical gradient
```

```
  #' is determined.
  #' @param h scalar. Finite difference value.
  #'
  #' @usage num_grad(f, x)
  #'
  #' @return Vector of partial derivatives of function f.

  n <- length(x)
  g <- rep(NA, n) # memory preallocation for storing gradient
  e <- diag(n)

  # calculation of partial derivatives
  for(i in 1 : n) g[i] = (f(x+h*e[i,])-f(x-h*e[i,]))/(2*h)
  return(g)
}
```

Listing 19: Numerical gradient implementation

The above function has comments that allow us to view the function documentation. It will contain a short description, input parameters and return values. We can access the documentation using the following command:

```
# Provides access to documentation
?num_grad
```

Listing 20: Function documentation access

After running this command, the following documentation will be displayed:

Central Numerical Gradient

Description

Function resposible for determining the numerical gradient by calculating the vector of central partial derivatives.

Usage

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

Arguments

f	function. The function which gradient we determine
x	numerical vector. The point at which the numerical gradient is determined
h	scalar. Finite difference value

Value

Vector of partial derivatives of function f.

Figure 5.13: Numerical gradient function docstring documentation

Now we can test how this numerical gradient function works. To accomplish this, first we need to prepare input parameters. Afterwards, we can invoke said function, check out its results and optionally benchmark it.

```r
# Input parameters
my_fun <- function(x) 2*x[1]^2 + x[2]^2
c(3,4) -> x0  # Caution! Arrows not only to the left!

# Results
my_fun(x) # 2*3^2+4^2 = 17 # Try also: sin(x^2);
x0 <- c(-3, -2)
my_fun(x0) # 2*(-3)^2+(-2)^2 = 22
num_grad(my_fun, x0, 10^-6) # returns vector [-12, -4]

# Benchmark 1
if(!require(numDeriv)) library(numDeriv)   # Library for numerical calculations
grad(my_fun, x0)    # gradient
hessian(my_fun, x0) # hessian

# Benchmark 2
if(!require(Deriv)) install.packages(Deriv); # Library for symbolic calculations
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 21: Numerical gradient tests

Having a working implementation of the gradient function at hand, we can move on to creating a function implementing the gradient descent algorithm:

```r
gradient_descent <- function(f, x, a = 0.1, K = 100){
  #' Gradient Descent
  #'
  #' @description Function responsible for calculating gradient descent
  #' of function f over K steps.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x numerical vector. The starting point for the algorithm.
  #' @param a scalar. Optional parameter that determines learning rate (defaults to 0.1).
  #' @param K scalar. Optional parameter that determines maximum iteration limit (defaults
    to 100).
  #'
  #' @usage gradient_descent(f, x, a, K)
  #'
  #' @returns
  #' List of various outputs containing the following:
  #' * x_opt: found solution,
  #' * f_opt: value of target function in the found solution,
  #' * x_hist: history of explored solutions,
  #' * f_hist: history of target function values,
  #' * t_eval: time elapsed during algorithm calculations.

  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){
    # description of the transition from point x_k to x_k+1
    x_new <- x - a * grad(f, x)

    # checking whether the new solution
```

```
    # is the best so far
    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
  }
  # time difference between the end and start of the algorithm
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 22: Gradient descent implementation

Example 14. Let $f(x_1, x_2) = \frac{1}{8}(x_1)^2 + (x_2)^2$ and $x_0 = (3, 4)$.

Function f at point x_0 has value equal to $17\frac{1}{8}$. After $K = 100$ iterations of our implementation of gradient descent algorithm, this value is worked down to approximately 0.00711 which is closer to global minimum of the function f - zero. Values at each step of this algorithm can be seen on the plot below:

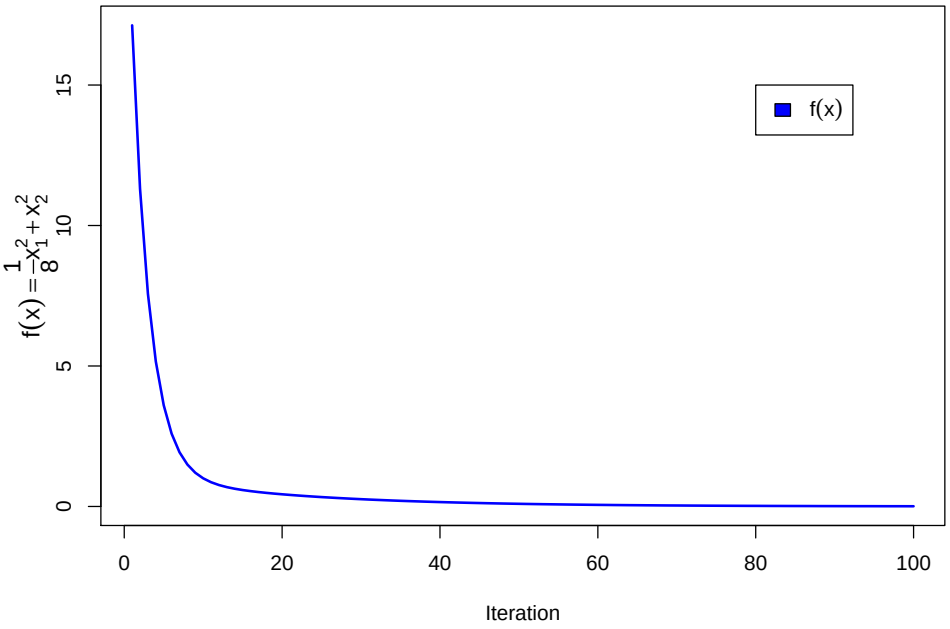


Figure 5.14: Values of function f gradually calculated over 100 iterations using gradient descent

We can plot not only values calculated by this algorithm, but also positions obtained in subsequent iterations in 2D space. The path covered by the gradient descent algorithm can be seen in the plot

below:

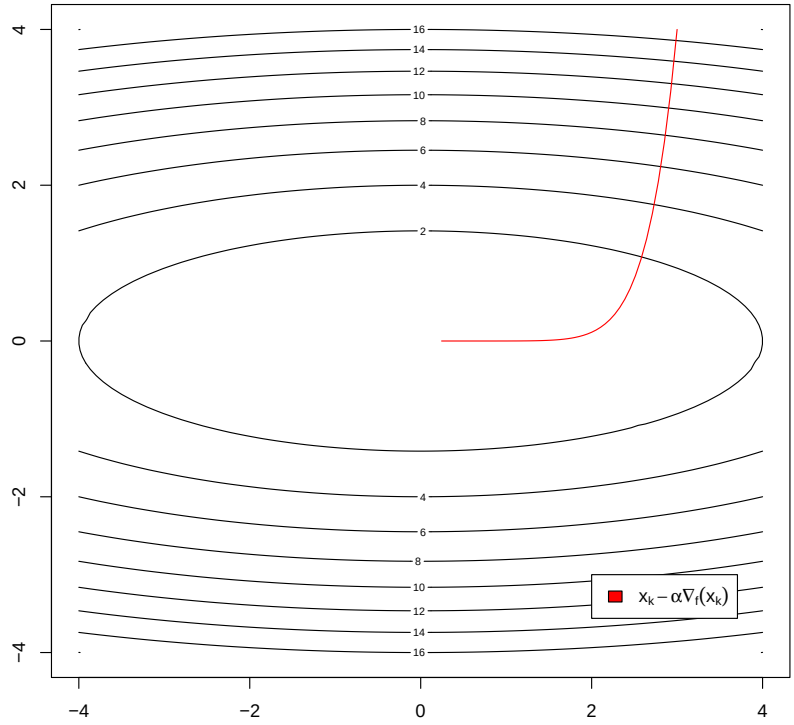


Figure 5.15: Path covered by the gradient descent algorithm on function f over 100 iterations in 2D space

5.16 Learning rate

The learning rate is a hyperparameter that plays a crucial role in controlling the step size at each iteration of the gradient descent algorithm. In the context of machine learning, learning rate is often represented by symbol α. The learning rate determines how much we should adjust the parameters with respect to the gradient of the cost function.

Why and how is it used? Lets consider following example:

Example 15. Knowing that the gradient indicates the greatest increase in function f, let $x_1 = x_0 - \nabla_f(x_0)$, where x_0 is a starting point and x_1 is the next point calculated in direction of biggest function decrease. For now assume that learning rate α is equal to 1.

Following the same process, we can calculate the points even further down the line in the iterative manner:

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

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

Let $f(x) = x^2$ and $x_0 = 1$. Knowing that the gradient of a 1D function is a simple derivative of said function, we can calculate points $x_0, x_1, x_2...$ for $f(x)$:

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

As we can see, a kind of deadlock has been achieved following such steps. Consecutive values just oscillate around the extremum, never converging to it.

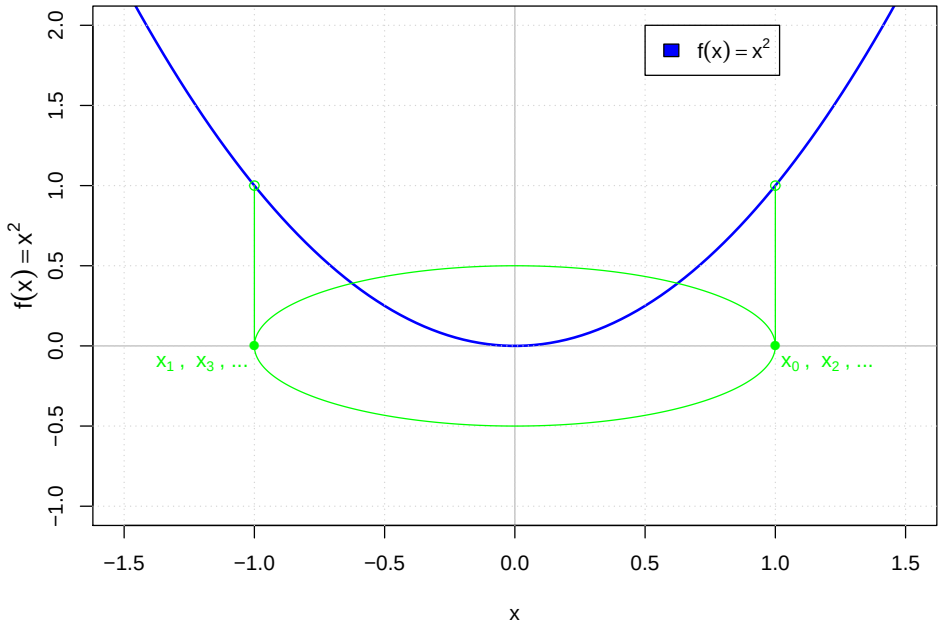


Figure 5.16: Simple deadlock created due to using whole gradient value while calculating points $x_0, x_1, x_2...$

The situation only gets worse if we assume that the function $f(x) = x^4$. When calculating points $x_0, x_1, x_2...$ we get increasingly larger values:

- $x_1 = x_0 - \nabla_f(x_0) = 1 - 4 = -3$
- $x_2 = x_1 - \nabla_f(x_1) = -3 - (-108) = 105$
- $x_3 = x_2 - \nabla_f(x_2) = 105 - 4,630,500 = -4,630,395$
- $x_4 = x_3 - \nabla_f(x_3) \approx -4630395 - (-3.9711301e + 20) \approx -3.9711301e + 20$ (...)

One of the possible solutions for problem illustrated in Example 4 is to introduce some kind of learning parameter, learning parameter or step size parameter (however we want to name it). This parameter would be responsible for controlling the size of the steps we are taking in each iteration. In other words, it would reduce the impact that gradient would have during each calculation. The mentioned parameter is exactly what learning rate is suppose to be. It is an integral part of the gradient descent algorithm and has big impact on how this algorithm performs.

The learning rate is a critical hyperparameter that needs to be carefully chosen. If the learning rate is too small, the algorithm may converge very slowly, requiring a large number of iterations to reach the minimum. On the other hand, if the learning rate is too large, the algorithm may overshoot the minimum and fail to converge or oscillate around the minimum.

Example 16. Let $f(x_1, x_2) = \frac{1}{8}(x_1)^2 + (x_2)^2$ and $x_0 = (3, 4)$.

The path covered by the gradient descent algorithm may vary depending on the chosen learnig rate. Let $\alpha_1 = 0.1, \alpha_2 = 0.4$ and $\alpha_3 = 0.8$. How gradient descent converges using this parameters can be seen on the plot below:

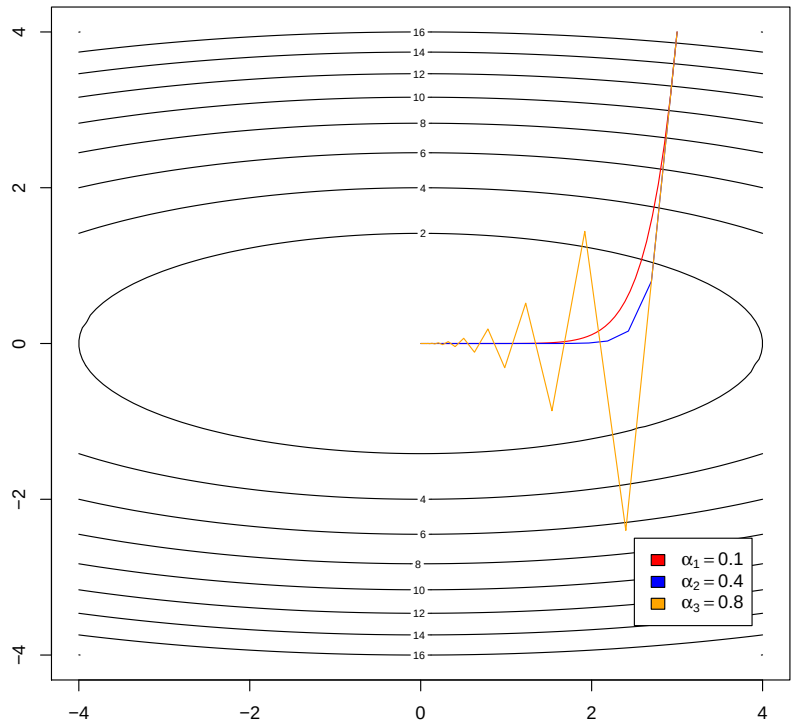


Figure 5.17: The impact of learning rate on the path traveled by the gradient descent algorithm on function f.

5.17 Gradient descent in neural networks

Gradient descent is often used when dealing with neural networks. It is a core optimization algorithm behind training of the neural network. The objective during training is to minimize a cost function, which measures the difference between the predicted outputs of the neural network and the actual target values.

Gradient descent is well-suited to use while optimizing multi-argument functions. Problems for which neural networks are used often deal with such functions. One of the common tasks for which neural networks are used is image classification.

When creating the neural network we need to specify its architecture, including the number of layers, the number of neurons in each layer, and the activation functions. The first layer of neurons, being the input layer, is directly related to the type of input data. In image classification, number of neurons in this layer is usually equal to number of pixels in analyzed image multiplied by 3 when dealing with colored input (for each of RGB channels).

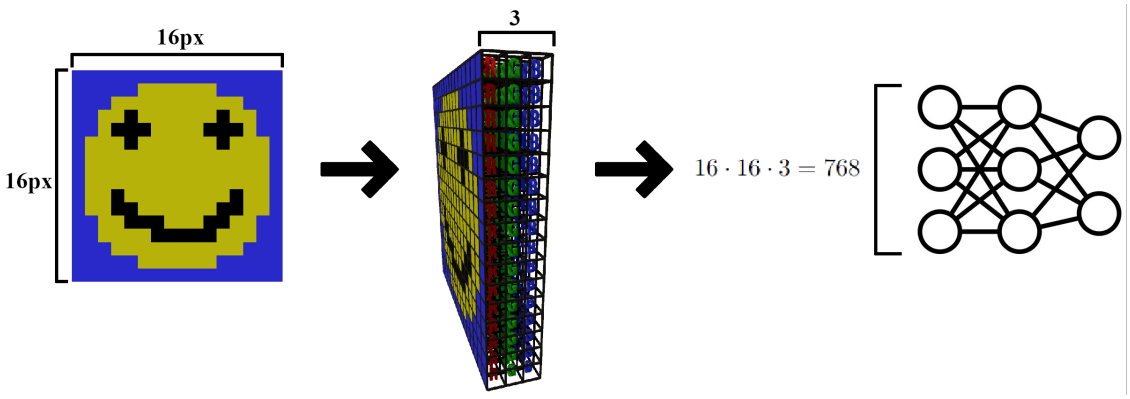


Figure 5.18: Number of pixels multiplied by 3 RGB channels as input for the neural network

In addition to the input layer, we also need to determine the number of neurons in the hidden layers and the output layer. The number and structure of hidden layers can vary depending on the chosen neural network approach. The number of neurons in the output layer however usually corresponds to the number of possible results - categories.

Each neuron in the network has connections with neighboring layers. This connections as well as neurons have special parameters called weights and biases. Weights represent the strength of connections between neurons in different layers of a neural network. Each connection between neurons is associated with a weight, and these weights are adjusted during the training process to enable the network to make accurate predictions. Biases are additional parameters in each neuron that allow the network to shift the activation function. They provide the model with flexibility to account for situations where the input to the neuron is insufficient to activate it. Biases are adjusted during training, along with weights, to improve the overall performance of the network. In a neural network layer, the output of each neuron is computed by applying a weighted sum of the inputs, followed by an activation function. Mathematically, the output O_j of neuron j in layer is given by:

$$O_j = \sigma\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right) \tag{5.43}$$

where w_{ij} is the weight of the connection between neuron i in the previous layer and neuron j in the current

layer, x_i is the input from neuron i, b_j is the bias of neuron j, σ is the activation function, and n is the number of neurons in the previous layer.

Weights and biases are chosen at random at first, but need to be adjusted during training of neural network. As one would expect, the results of randomly initialized networks are not good in most cases. To estimate how well a neural network performs, a so-called *costfunction* is created. It is a mathematical measure that quantifies the difference between the predicted output of the network and the actual target values. Common cost function used for regression problems, where the goal is to predict a continuous value, is called Mean Squared Error (MSE). The mean squared error is the average of the squared differences between the predicted and actual values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.44}$$

where n is the number of data points, y_i is the actual target, and $\hat{y}_i$ is the predicted output.

We can think of the cost function as a function that accepts all network weights and biases as input, and outputs one number that is an assessment of the network's performance. This multi-parameter function is then minimized using gradient descent. The vector created using gradient descent contains a number of changes that should occur in each of the weights and biases to approach the minimum of the cost function - so that the results obtained by the network are closer to the expected results.

$$\nabla C(w_0, w_1, ..., w_n) \tag{5.45}$$

where ∇C is gradient used in gradient descent, n is combined number of all the weights and biases and w_n is value of n-th weight or bias.

5.18 Steepest descent

Steepest descent is another optimization algorithm commonly used to minimize nonlinear analytical functions. It is very similar to the gradient descent algorithm in its structure. The goal of steepest descent is also to iteratively move towards the minimum of a function by adjusting its parameters. However, in contrary to gradient descent, we do not set the learning rate parameter of the algorithm upfront. We only provide the maximum learning rate value (maximum step size in direction of antigradient), and then the algorithm determines the optimal value of this parameter.

Definition 15: Steepest descent

Given that steepest descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - \alpha_{k-best}\nabla_f(x_k) \tag{5.46}$$

where $k \in \mathbb{N}$ is an iteration number, $\nabla_f(x_k)$ is the gradient of a function f in point x_k and α_{k-best} is the best learning rate found over g steps of linear search.

Using this algorithm, we are not only taking steps in the optimal direction, but also automatically select the best size of those steps. We can calculate the best step size or learning rate in several ways. One way is

to take advantage of the gradient descent formula and use it to calculate the its derivative over the learning rate α. Finding the minimum of said derivative would give us the optimal α parameter.

$$x_{k+1} = \underset{\alpha}{\operatorname{argmin}} \frac{d}{d\alpha} \left(x_k - \alpha \nabla_f(x_k)\right) \tag{5.47}$$

Unfortunately, there is a problem with this solution. If we were to use derivative in our equation, we would end up with symbolical algorithm. Implementing such an algorithm would require us to be able to quickly calculate the derivative of any function. This is not always an easy task. It is easier for computers to perform calculations using numerical algorithms. This leads us to the next solution - approximating the optimal learning rate value.

To approximate step size we can linearly search through different points along the direction pointed by the antigradient and check what results we get with them. Of course, we don't want to search through the infinite number of points we can find on this line. For this purpose, we should set the maximum search range (i.e. the upper α limit - α_{max}) as well as the starting point of the search x_k. However, there are also an infinite number of points on the section. Therefore, we also set the g parameter, which determines how many points we should check on this section.

The steepest descent algorithm resembles the gradient descent algorithm with one key difference. In each step of the algorithm, we determine g points in the direction of the antigradient and arrange them at an equal distance from each other on a section of length controlled by α_{max}. It is equivalent to discreetly going over g steps from $\alpha = 0$ to $\alpha = \alpha_{max}$ and choosing the best outcome.

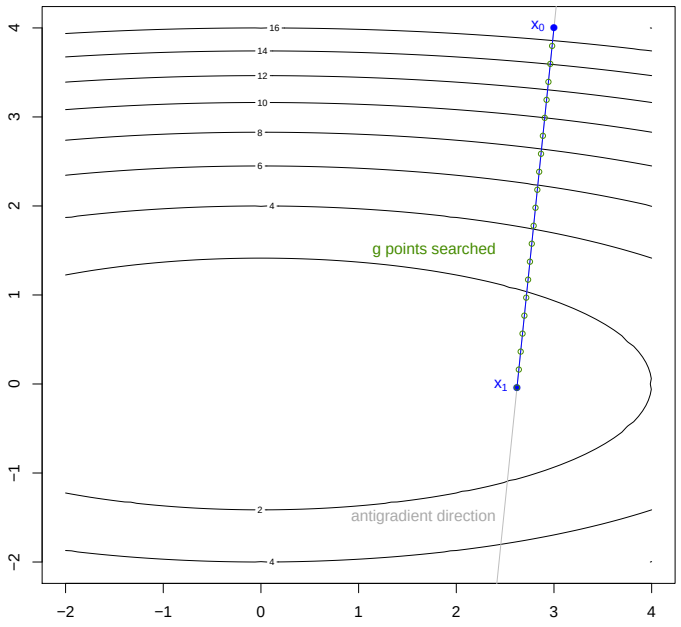


Figure 5.19: Points considered during one iteration of steepest descent algorithm of function f

This is somewhat of a heuristic approach. This algorithm will not provide us with optimal value of the step

size but an approximation of one. With a sufficiently large parameter g, we will obtain a value close to the optimal at a relatively low computational cost. The main advantage of this algorithm compared to the gradient descent algorithm is a better adjusted step size in each iteration. Unfortunately, this also has disadvantages - including higher computational costs of a single iteration, which, however, is often offset in the long run by the better quality of the obtained steps.

Example 17. Let $f(x_1, x_2) = \frac{1}{8} \cdot (x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

Additionally, let the parameters of steepest descent algorithm be $\alpha_{max} = 5$ and the parameter $g = 1000$. How steepest descent converges using this parameters can be seen on the plot below (alongside with path created by gradient descent):

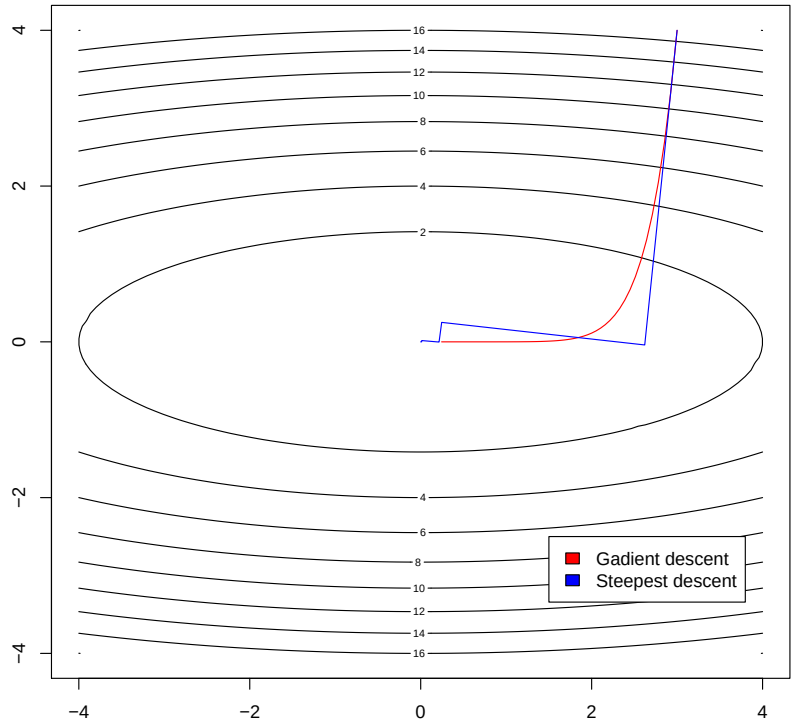


Figure 5.20: Comparison of the path obtained by the steepest descent and gradient descent algorithms on function f in 2D space

We can observe an interesting pattern in the path created by the steepest descent algorithm. The consecutive steps of the algorithm iterations create sections perpendicular to each other. The same pattern repeats itself and gets closer and closer to the extremum of the function. This behavior is not random or accidental. This is due to the fact that the gradient of the function at any point does not have to indicate the global extremum, but the direction of the fastest decline of the function value. The step size is adjusted linearly until the function stops decreasing and reaches stationarity. It would then reach the tangent space, where the function stops decreasing (and as it moves further, it starts increasing). From the newly determined point, since it lies on tangent space, the direction

of the function's fastest decrease is at an angle of 90^o from the direction of the last step.

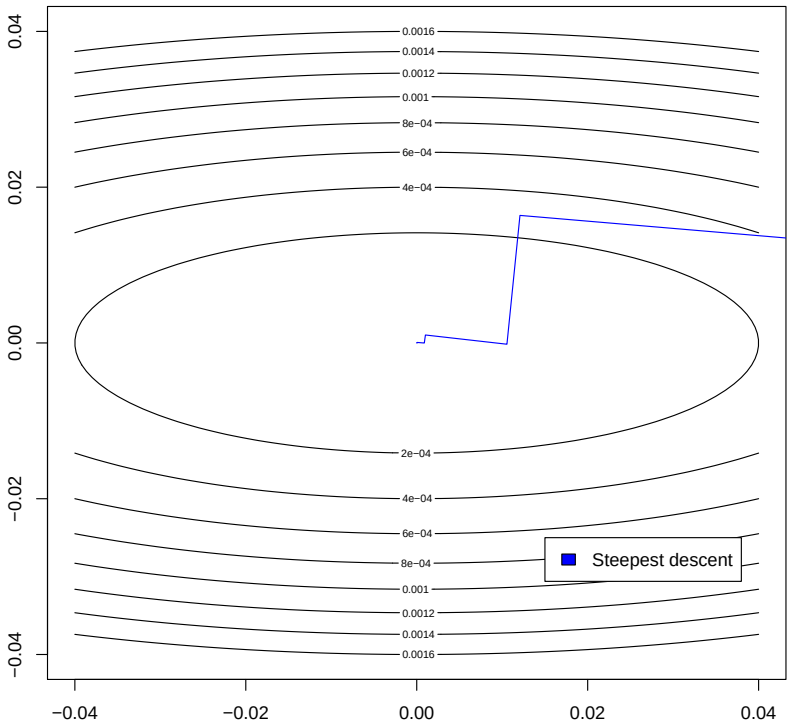


Figure 5.21: Closeup of the path obtained by the steepest descent algorithm on function f in 2D space

Using programming language **R** we can write a steepest descent implementation. First of all, we need a function to linearly search the best solution over g points:

```
line_search <- function(f, x0, x1, g = 100) {
  #' Line search
  #'
  #' @description Helping function responsible for finding the best point judging by
    function f
  #' over g linearly distributed points.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x0 numerical vector. The starting point for the algorithm.
  #' @param x1 numerical vector. The end point for the algorithm (or max range of step).
  #' @param g scalar. Optional parameter that is responsible for the number of iterations of
     searching for the best step size
  #' in each iteration of the algorithm itself (defaults to 100).
  #'
  #' @usage line_search(f, x0, x1, g)
  #'
  #' @returns
  #' x_best: best found point judging by function f

    # setting x0 as a starting point
```

```
    x_best <- x0
    # looping over g points in direction of point 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 23: Line search implementation

Heaving the line search function ready, we can proceed to implement the main part of the algorithm. It can be done as a following function *steepest descent* that is based on previous gradient descent implementation:

```
steepest_descent <- function(f, x, a = 5, g = 100, K = 100){
  #' Steepest Descent
  #'
  #' @description Function responsible for calculating steepest descent
  #' of function f in g points each iteration over K steps.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x numerical vector. The starting point for the algorithm.
  #' @param a scalar. Optional parameter that determines max learning rate (defaults to 5).
  #' @param g scalar. Optional parameter that is responsible for the number of iterations of
     searching for the best step size
  #' in each iteration of the algorithm itself (defaults to 100).
  #' @param K scalar. Optional parameter that determines maximum iteration limit (defaults
    to 100).
  #'
  #' @usage steepest_descent(f, x, a, g, K)
  #'
  #' @returns
  #' List of various outputs containing the following:
  #' * x_opt: found solution,
  #' * f_opt: value of target function in the found solution,
  #' * x_hist: history of explored solutions,
  #' * f_hist: history of target function values,
  #' * t_eval: time elapsed during algorithm calculations.

  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){
    # description of the transition from point x_k to x_k+1
    x_new <- line_search(f, x, x - a * grad(f, x), g)

    # checking whether the new solution
    # is the best so far
    if(f(x_new) < results$f_opt){
      results$x_opt <- x_new
      results$f_opt <- f(x_new)
    }
```

```
44
45      results$x_hist[k,] <- x_new
46      results$f_hist[k] <- f(x_new)
47
48      x <- x_new
49    }
50    # time difference between the end and start of the algorithm
51    results$t_eval <- Sys.time() - start_time
52    return(results)
53  }
```

Listing 24: Steepest descent implementation

131890-S Optimization methods **Winter 2024/2025**

Lecture 6: More local methods

Daniel Kaszyński

6.19 Newton Descent

6.19.1 Theory behind Newton Descent

Newton descent is another iterative algorithm often used to minimize functions, similarly to gradient descent and steepest descent discussed in previous lectures. It uses second order approximations to find critical points of given function f. The basic idea of Newton's method in optimization is to iteratively update an initial guess for the optimal solution based on the function's first and second derivatives. The update rule is derived from the Taylor series expansion of the function around the current guess.

To better understand how this algorithm works, let's start with Taylor series approximation around point x_k going up to second degree derivative (second-order Taylor approximation of f):

$$f(x_k + h) \approx f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2 \tag{6.48}$$

The goal of this method is to find h for which the function around a given point x_k changes the fastest. We assume that x_k in each step is given and therefore constant. With this information we can make an observation that Taylor series is a poly-nominal formula for approximating values ($f(x_k)$, $f'(x_k)$ and $\frac{1}{2}f''(x_k)$ are constant leaving us with only h to work with).

Next up, we can apply First Order Conditions and equate the derivative of our approximation to zero:

$$\frac{d}{dh}\left(f(x_k) + f'(x_k)h + \frac{1}{2}f''(x_k)h^2\right) = f'(x_k) + f''(x_k)h = 0 \tag{6.49}$$

Then, using a simple formula transformation, we can obtain the formula for the optimal value of h:

$$h = -\frac{f'(x_k)}{f''(x_k)} \tag{6.50}$$

This formula for h can be used to calculate steps taken in subsequent iterations of Newton descent algorithm for 1D functions.

Definition 16: Newton Descent for 1D function

Given that Newton descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k + h = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{6.51}$$

where $k \in \mathbb{N}$ is an iteration number, $f'(x_k)$ is the derivative of a function f in point x_k and $f''(x_k)$ is the second derivative of a function f in point x_k.

The Newton descent presented in formula above (6.51) can be generalized to more than one dimension by replacing the derivative with the gradient and the reciprocal of the second derivative with the inverse of the Hessian matrix (because as we know from previous lectures, Hessian matrix is a generalization of a second derivative).

Definition 17: Newton Descent

Given that Newton descent is an iterative algorithm, by a point calculated at $(k-1)$-th step of the gradient descent of function $f : \mathbb{D} \subset \mathbb{R}^n \to \mathbb{R}$, $x_k \in \mathbb{D}$ we call:

$$x_{k+1} = x_k - H_f(x_k)^{-1} \nabla_f(x_k) \tag{6.52}$$

where $k \in \mathbb{N}$ is an iteration number, $H_f(x_k)^{-1}$ is the inverse of the Hessian matrix and $\nabla_f(x_k)$ is the gradient of a function f calculated in point x_k.

Caution! When using Newton descent we do not know by default if we are optimizing functions towards minimum, maximum or other critical points of given function. Newton descent is steering towards stationarity point of a function. That's why it is good practice to use Newton descent as the last optimization step in order to more quickly reach the extremum of the function that we approached using another algorithm.

Example 18. Let $f(x) = ax^2 + bx + c$ and x_0 a starting point for Newton descent algorithm.

Using Newton descent formula for the given 1D function f, we get:

$$x_1 = x_0 - \frac{2ax_0 + b}{2a} \tag{6.53}$$

We can then divide the fraction obtained on the right side of the equation into two parts:

$$x_1 = x_0 - \frac{2ax_0}{2a} - \frac{b}{2a} \tag{6.54}$$

Thanks to this transformation, we can eliminate expressions containing x_0:

$$x_1 = -\frac{b}{2a} \tag{6.55}$$

The obtained formula for x_1 is the formula for a vertex of second order polynomial function (in this case minimum). So in one step we are able to find the extremum of a function. As we can see, this algorithm is very efficient whenever we face problems dealing with quadratic forms.

As we can see in the formula for Newton descent (6.52), to calculate the subsequent steps taken in each iteration of the algorithm, we must be able to get the Hessian of the function. To be able to calculate the Hessian of a function one must derive a partial derivative over x_i and x_j.

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

To calculate the Hessian numerically, we need to use approximations of partial derivatives - finite differences. They are denoted using the Δ character. Therefore, the Hessian matrix using said approximations would

look like this:

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

Keeping the formula for central difference in mind, we can write the single finite difference over x_i as:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{\Delta f}{\Delta x_i}(x) = \frac{f(x + e_i h) - f(x - e_i h)}{2h} \tag{6.56}$$

Similarly, each entry of the Hessian matrix, being finite difference over x_i and x_j, would follow this formula:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j}(x) = \frac{f(x + e_i h + e_j h) - f(x + e_i h - e_j h) - f(x - e_i h + e_j h) + f(x - e_i h - e_j h)}{4h^2} \tag{6.57}$$

It is worth noting that the order in which approximations of partial derivatives are applied is not important. If function $f \in C^2$ (it is at least twice-differentiable), using Schwartz theorem we can prove that:

$$\frac{\Delta^2 f}{\Delta x_i \Delta x_j} = \frac{\Delta^2 f}{\Delta x_j \Delta x_i} \tag{6.58}$$

which also implies that:

$$H_f(x) = H_f^T(x) \tag{6.59}$$

Another thing to remember is that not from every function Hessian is easily obtainable (for example functions that do not have curvature). To ensure correct flow of work of the algorithm, a certain trick must be used. This trick comes down to adjusting Hessian of a function by adding to it diagonal matrix multiplied by small λ value. This change should be applied whenever determinant of $abs(H_f(x))$ is smaller that some arbitrary threshold t. Described method is often called **Ridge Regression** or **Levenberg-Marquardt adjustment**.

6.19.2 Newton-Raphson method

There is also another algorithm, similar to Newton descent, that is worth mentioning. It is called the Newton-Raphson method (and sometimes just Newton's method). This algorithm is a root-finding algorithm which produces successively better approximations to the roots (or zeroes) of a given function. The formula used by this algorithm is as follows:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{6.60}$$

Caution! This is a very similar formula to the formula used by Newton descent, but it differs in the degree of derivatives of the function f. The two mentioned algorithms are trying to solve two different problems. However, they share a common relation. The Newton-Raphson method finds the roots of the function that will be given to it. Assuming that a given function is already a derivative of another function (considered by the Newton descent algorithm), finding these roots basically gives us the formula for Newton descent and solves the problems it addresses.

6.19.3 Implementation of the Newton descent algorithm

Using programming language **R** we can write a Newton descent implementation. First of all, we need a function to calculate numerical Hessian matrix:

```
num_hessian <- function(f, x, h = 10^-3){
  #' Numerical Hessian
  #'
  #' @description Function responsible for determining the numerical Hessian
  #' by calculating the matrix of partial derivatives.
  #'
  #' @param f function. The function which Hessian we determine
  #' @param x numerical vector. The point at which the numerical Hessian
  #' is determined
  #' @param h scalar. Finite difference value
  #'
  #' @usage num_hessian(f, x)
  #'
  #' @return Hessian matrix of partial derivatives of function f.

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

  for(i in 1 : n) { # Rows
      for(j in 1 : n) { # Columns
          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 25: Numerical Hessian implementation

The formula used in subsequent iterations of the Newton descent algorithm requires us to calculate the inverse of the Hessian matrix. Fortunately, language **R** provides a ready-made solution to this problem in the form of the *solve()* method.

```
my_fun <- function(x) 1/8*x[1]^2+x[2]^2
x0 <- c(3, 4)
solve(num_hessian(my_fun, x0)) # Returns the inverse of the Hessian matrix
```

Listing 26: Hessian solve() method example

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:

```
newton_descent <- function(f, x, K = 100){
  #' Newton descent
  #'
  #' @description Function performing the Newton descent algorithm.
  #'
  #' @param f: objective function
  #' @param x: intial point
  #' @param K: maximum number of iterations
  #'
  #' @returns
  #' List that contains following elements:
  #' * x_opt: optimal solution
```

```
  #' * f_opt: objective function value of optimal solution
  #' * x_hist: history of explored points
  #' * f_hist: history of objective function values
  #' * t_eval: running time of algorithm

  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){
    # calculating gradient and hessian of a function f
    G <- grad(f, x)
    H <- num_hessian(f, x)

    # using Ridge Regression to help solve situations
    # where Hessian can not be calculated
    if(abs(det(H)) < 10^(-3)) H <- H + diag(n)*10^(-3)

    # description of the transition from point x_k to x_k+1
    x_new <- x - solve(H) %*% G

    # checking whether the new solution
    # is the best so far
    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
  }
  # time difference between the end and start of the algorithm
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 27: Example Newton descent implementation

Example 19. Let $f(x_1, x_2) = \frac{1}{8}(x_1)^2 + (x_2)^2$ and $x_{start} = (3, 4)$.

How Newton descent converges can be seen on the plot below (alongside with paths created by other similar algorithms):

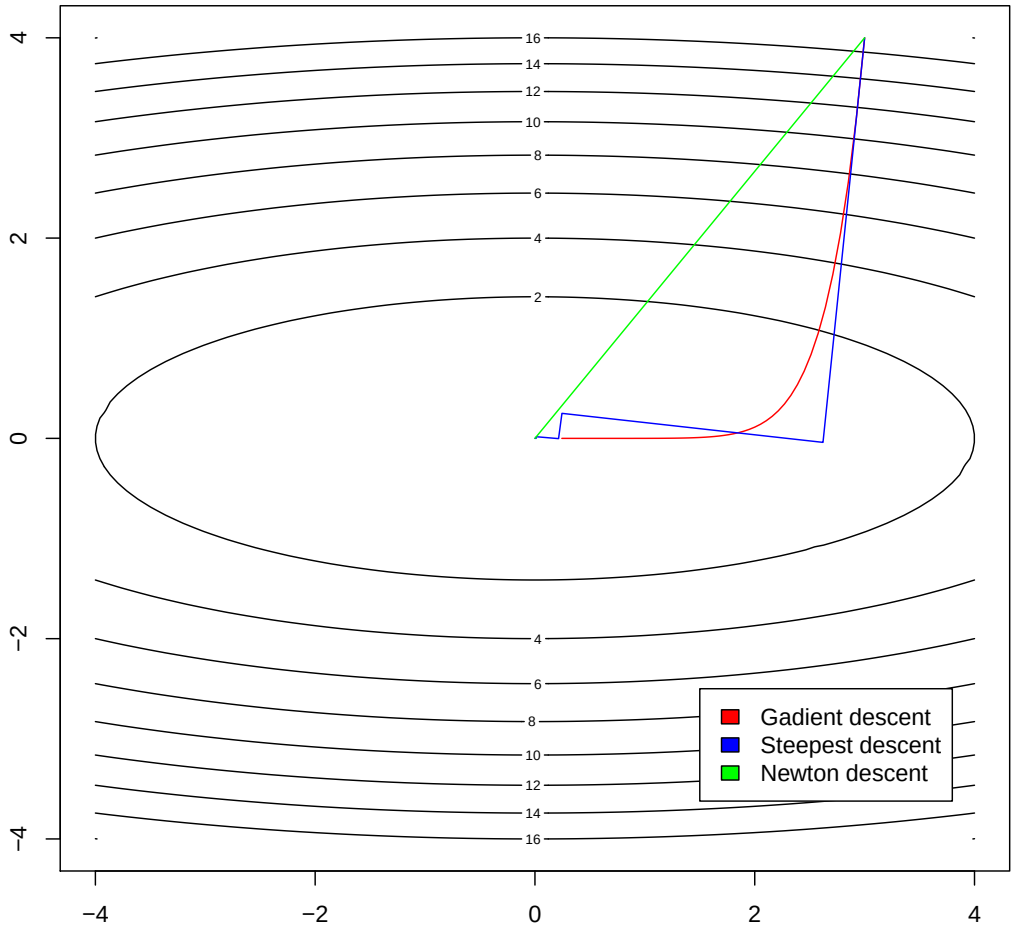


Figure 6.22: Comparison of the paths obtained by the Newton descent and other similar algorithms on function f in 2D space

We can see that Newton descent has a path that is significantly different from the other algorithms. Steepest descent and gradient descent algorithms are moving orthogonal with regards to counter-plot. Newton descent takes a much more direct route towards the global minimum of the function. It is no longer going towards locally best direction. This behavior is caused by the fact that the function f is quadratic.

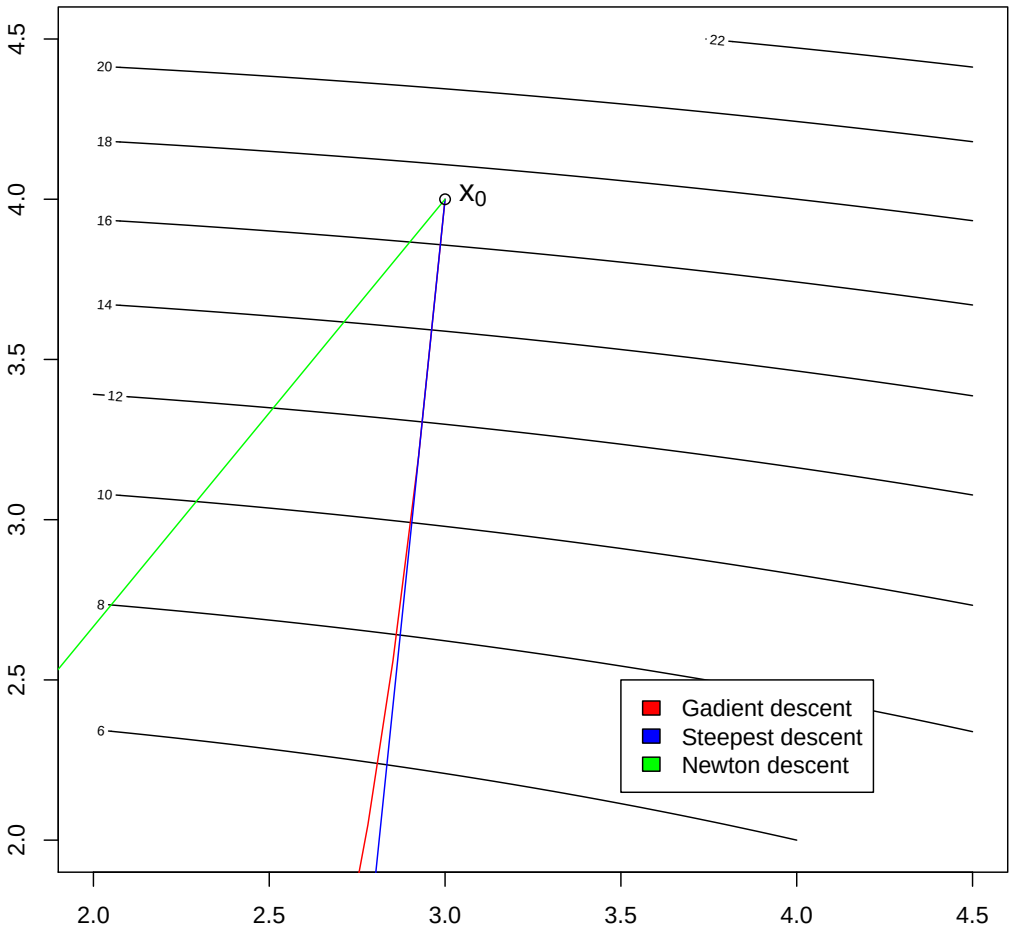


Figure 6.23: Closeup of the paths obtained by the Newton descent and other similar algorithms

Lecture 7: Simulated Annealing

Daniel Kaszyński

7.20 Schaffer function

When working on optimization methods, it is a common practice to thoroughly test the algorithms we create. Simple functions only provide us with the opportunity to evaluate our solutions to a certain extent. If we want to assess the characteristics of optimization methods in more difficult cases, it is worth using more complex functions.

There are many complex functions on which complicated optimization algorithms can be tested. The list of sample test functions can be found, among others, on Wikipedia website.

One of the popular functions for testing optimization algorithms is the Schaffer function. It is expressed using the following formula:

$$f(x_1, x_2) = 0.5 + \frac{sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 + y^2)]^2} \tag{7.61}$$

This function has a minimum at the point $f(0, 0) = 0$ and has values in the range $< 0, 1 >$.

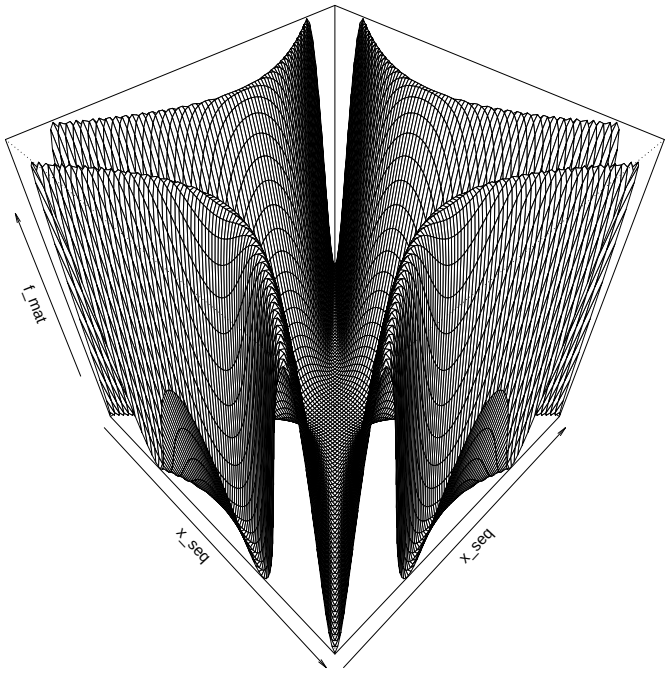


Figure 7.24: Schaffer function plot in 3D space.

On the plot of the function we can see that near the point $(0, 0)$ there is a valley containing the global extremum of the function. This valley takes the shape of an "X" and is surrounded by drastic jumps in the function values. The space behind these fluctuations is characterized by the occurrence of dips and rises in the function values in a wave-like pattern. It is also worth noting that the wave contours lie almost parallel to each other.

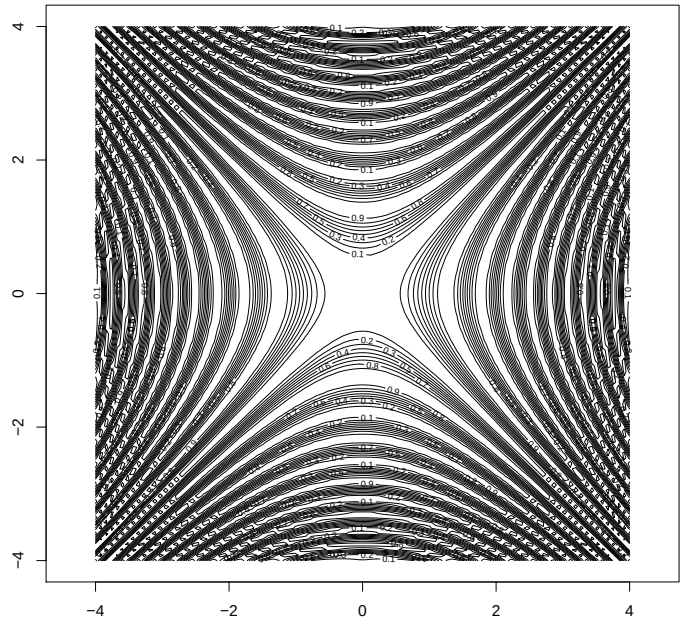


Figure 7.25: Contour plot of Schaffer function.

This function is great for testing the ability of algorithms to explore different sections of the function space in search of the global extremum. The Schaffer function has a diverse topography. Without proper exploration, algorithms may get stuck in a local extremum. This may lead to suboptimal solutions that are not globally best.

To avoid getting stuck in a local extremum, several different exploration techniques can be used. These include, among others, stochastic methods, i.e. algorithms based on randomness, such as evolutionary algorithms or genetic algorithms. Other methods may be multi-point methods, population methods or adaptive exploration strategies.

7.21 Tests of local search algorithms

To begin with, it is worth determining how the optimization algorithms we learned in previous lectures, such as the gradient descent algorithm or the steepest descent method, behave on the Schaffer function.

Example 20. Let function f be the Schaffer function and starting point of the extremum search $x_{start} = (3, 4)$.

Moreover, let's assume the following values as parameters of the gradient descent: learning rate $\alpha = 0.02$ and maximum iteration limit $K = 30000$. Similarly, in steepest descent method, assume that maximum learning rate $\alpha = 5$, number of iterations to search for the best learning rate $g = 1000$ and maximum iteration limit $K = 30000$.

When attempting to optimize the function from the starting point x_{start} using the gradient descent algorithm and the steepest descent method, both of them approached the local extremum of the function. The paths obtained in the subsequent iterations of these algorithms can be seen on the plot below:

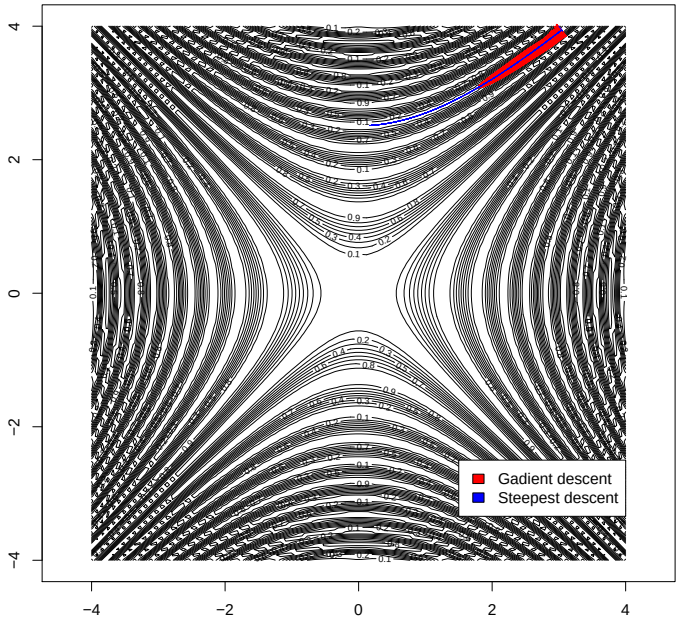


Figure 7.26: The paths obtained by gradient descent and steepest descent algorithms on Schaffer function.

It is easy to notice that none of the algorithms managed to get out of the initial valley near the starting point x_{start}. They moved very slowly and required a large number of iterations. Moreover, they became stuck pursuing the local minimum of the function. This phenomenon is known as the local extremum trap. Based on this knowledge, it can be concluded that gradient descent and steepest descent do not have global search properties. In the case of functions such as the Schaffer function, it is worth introducing escape mechanisms that allow algorithms to searching other regions.

7.22 Simulated Annealing

Simulated annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. In metallurgy, annealing is a technique used to reduce defects and improve the crystal structure of materials by heating them and then gradually cooling them. Similarly, in the context of optimization, simulated annealing is used to find the global minimum of a function by imitating a stepwise cooling process.

The first step of the simulated annealing algorithm is its initialization. In this phase, the starting parameters of the algorithm are set, which include:

- f - considered target function,
- x_0 - starting point of the algorithm,
- d - neighbourhood,
- t_0 - starting temperature,
- α - temperature drop rate,
- K - number of iterations of the algorithm,

and internal parameters such as:

- A_k - value of the activation function,
- t_k - temperature in each iteration.

Simulated annealing uses a temperature parameter that controls the likelihood of worse solutions being chosen as the algorithm progresses. Initially, the temperature is set to a high value and is gradually reduced in subsequent iterations, in accordance with the adopted annealing strategy.

In each iteration of the algorithm, an neighbor solution is generated. This is a potential candidate obtained by making a small random change to the current solution within neighborhood d. This is followed by an evaluation of the target function for the current solution and the neighboring solution. If the neighboring solution is better (i.e. has a lower target function value), it is accepted as the new, current solution. If the neighbor's solution is worse, it can be accepted with a probability given by the activation function A_k and the current temperature t_k. This activation (or probability) function allows the algorithm to sometimes accept worse solutions at the beginning of the optimization process, which helps prevent the algorithm from getting stuck in local minima. Then the temperature is lowered in accordance with the adopted strategy. As the temperature decreases, the probability of adopting worse solutions decreases and the probability of the algorithm convergence towards the global minimum increases. These steps are repeated in subsequent iterations until the stopping criterion is met, which may be reaching the maximum number of iterations or reaching a satisfactory solution.

Simulated annealing is effective in finding near-optimal solutions to complex optimization problems where traditional gradient-based methods can be problematic, especially when the objective function is non-convex or noisy. By allowing the algorithm to occasionally accept inferior solutions, simulated annealing can explore a wider range of the solution space and avoid getting trapped in local minima.

Using the **R** programming language, we can write our own implementation of the *simulated annealing* algorithm. An example implementation of this method might look like this:

```
simulated_annealing <- function(f, x0, d, t0, a, K = 100){
  #' Simulated Annealing
  #'
  #' @description The function responsible for approximating the global extremum
  #' for the function f in K steps using the simulated annealing algorithm.
  #'
  #' @param f function. The target function for the algorithm.
  #' @param x0 numerical vector. The starting point for the algorithm.
  #' @param d scalar. Considered neighborhood.
  #' @param t0 scalar. Initial temperature.
  #' @param a scalar. A parameter that determines the rate of temperature decline.
  #' @param K scalar. Optional parameter that determines maximum iteration limit (defaults
    to 100).
```

```
  #'
  #' @usage simulated_annealing(f, x0, d, t0, a, K)
  #'
  #' @returns
  #' List of various outputs containing the following:
  #' * x_opt: found solution,
  #' * f_opt: value of target function in the found solution,
  #' * x_hist: history of explored solutions,
  #' * f_hist: history of target function values,
  #' * t_eval: time elapsed during algorithm calculations.

  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){
    # choice of a next solution from the neighborhood
    x_c <- x + runif(n, min = -d, max = d)
    A_k <- min(1, exp(-(f(x_c) - f(x)) / (t_k)))

    # checking whether the new solution
    # should be accepted as new
    # and if it is the best so far
    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
  }

  # time difference between the end and start of the algorithm
  results$t_eval <- Sys.time() - start_time
  return(results)
}
```

Listing 28: *Simulated annealing* algorithm implementation

It is worth noting that the *runif()* function was used to select the neighboring solution from the neighborhood. This function provides random variations according to a normal distribution within the specified range from *min* to *max*.

Example 21. Let function f be the Schaffer function and starting point of the extremum search $x_{start} = (3, 4)$.

Moreover, let's assume the same values as in the previous example for the gradient descent and steepest descent algorithms , except that the maximum iteration is reduced to 2000. Let us also assume that the *simulated annealing* algorithm takes the parameters $d = 0.8$, $t_0 = 100$, $\alpha = 0.99$ and the maximum iteration limit $K = 2000$.

When trying to optimize the function from the starting point x_{start} using the *simulated annealing* algorithm, we can see that it explores different regions of the considered function. It does not stop at local minima and even leaves the local extremum in order to investigate further values, that have potential to be global extremum. The paths obtained in the subsequent iteration steps of these algorithms can be seen in the plot below:

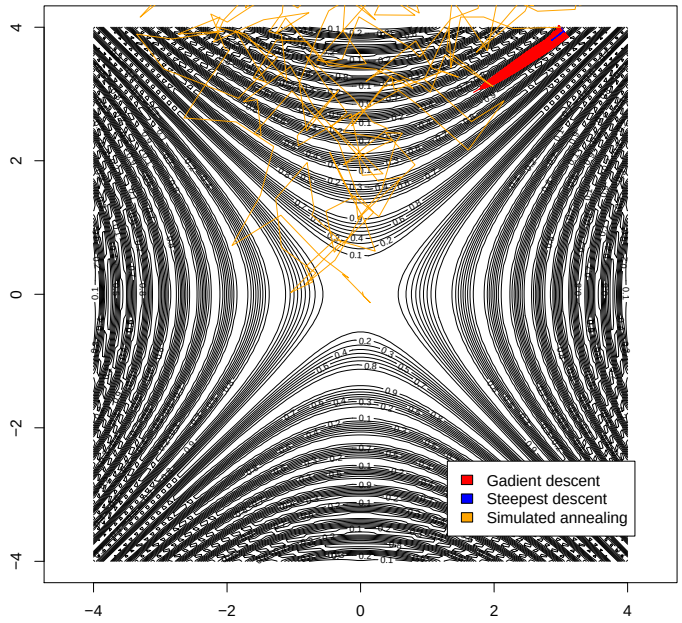


Figure 7.27: Path created by *simulated annealing* algorithm in comparison to gradient descent and steepest descent on Schaffer function.

During subsequent iterations of the algorithm, the value of the temperature parameter t_k gradually decreases. This reduces the randomness that occurs in the algorithm's search behavior and focuses it on the best solution found so far. Over time, the algorithm stops focusing on exploring new values and tries to optimize the best existing solution. The temperature drop in subsequent iterations of the algorithm can be seen in the plot below:

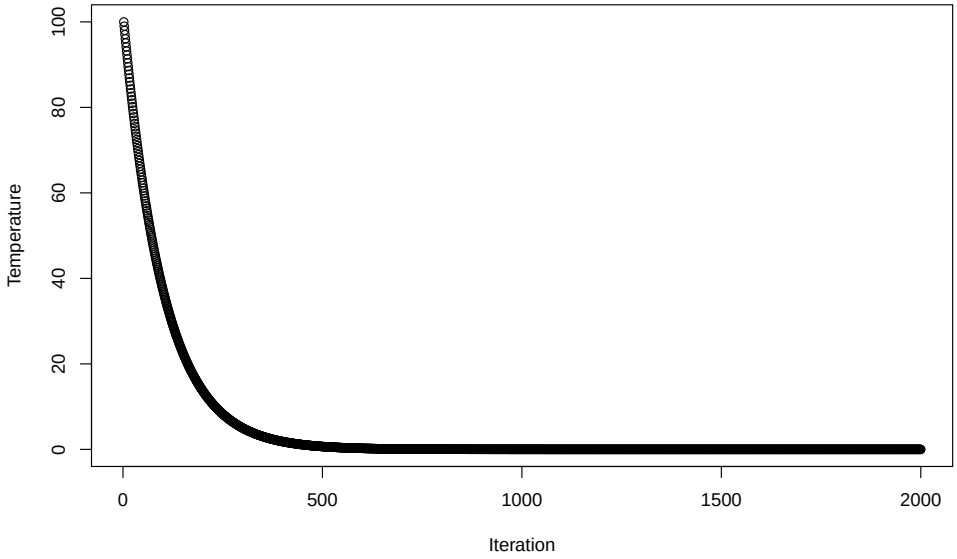


Figure 7.28: Plot of temperature values over subsequent iterations of the algorithm.

It is also worth taking a look at the values of the activation function in each of iterations of the algorithm. They decrease over the course of the algorithm's operation from values close to 1 to values close to 0. We can see the values of the activation function in the plot below:

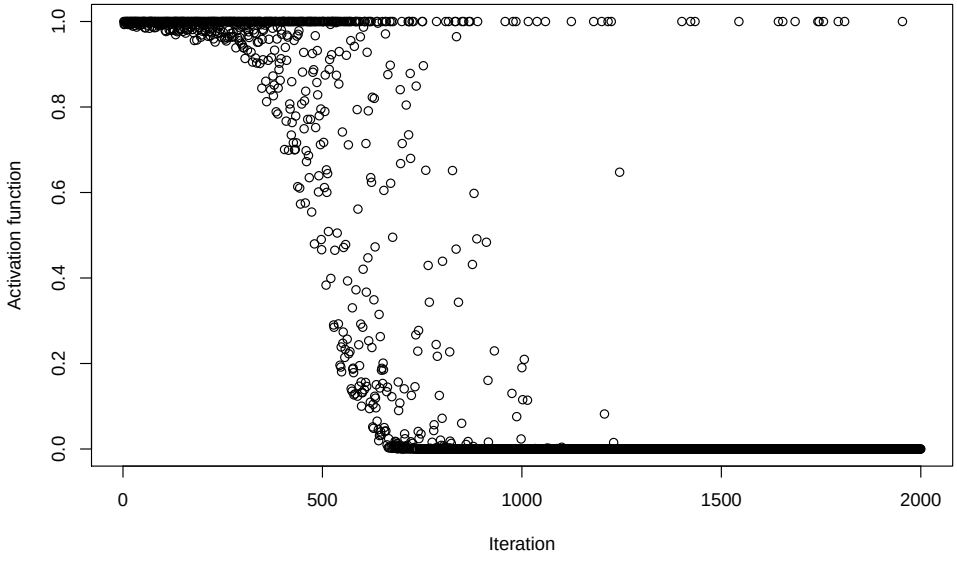


Figure 7.29: A plot of the activation function values over subsequent iterations of the algorithm.

When searching for a solution, we can distinguish three phases. They are easy to observe on the activation function plot. At the beginning, the activation function has large values (approximately up to 200-300 iterations). This means that the algorithm often accepts considered candidates for the optimal x, even if the values of the target function are not desirable in their case. Then comes the second phase, in which good solutions have a high probability of acceptance and poor solutions have a low probability of acceptance (approximately up to 1000-1200 iterations). The last phase is the phase in which the behavior of the switch model is adopted - better solutions take the value of 1, and worse ones take 0. High values occur much less often because it is harder to find a better solution the longer the algorithm runs, because there is less chance of improving the result.

The mentioned three phases that are characteristic to this algorithm are also reflected by values of the target function obtained during the operation of the algorithm. A plot showing these values is provided below:

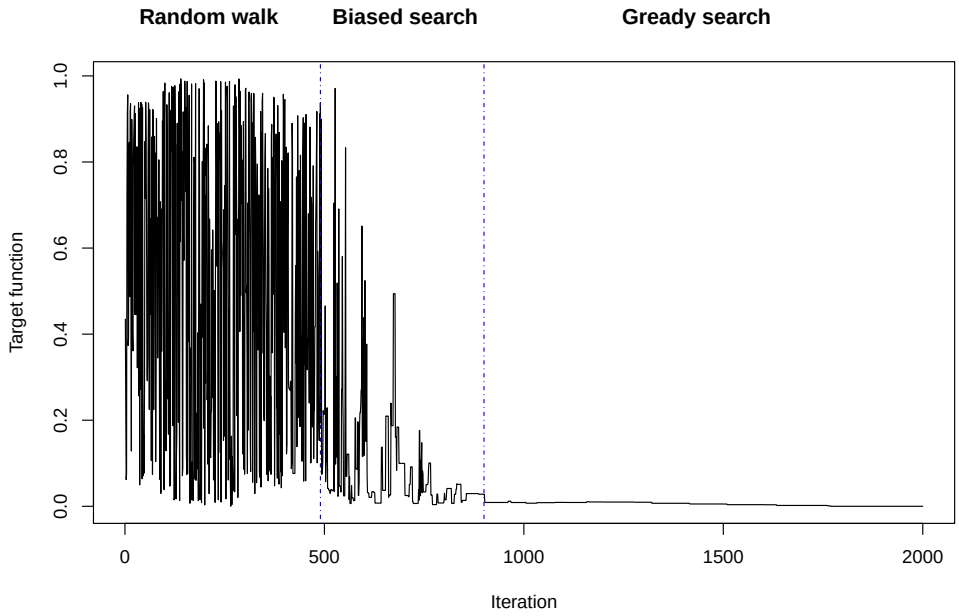


Figure 7.30: Plot of the targt function values over subsequent iterations of the algorithm.

Similarly as in the case of the activation function, three phases of the algorithm's operation are also visible. At the beginning, the values of the target function assume values scattered from the minimum to the maximum of the function, because almost all solutions encountered were accepted. Then the frequency and intensity of jumps in function value decreased and were no longer so drastic. In the final phase, the target function values stabilized and were slowly reduced.

The *simulated annealing* algorithm is very sensitive to the definition of the neighborhood. It's accuracy depends largely on the correct selection of starting parameters. Furthermore, it suffers from the curse of dimensionality present in optimization problems. As the number of dimensions of the functions we work on increase, the probability of choosing the correct direction when analyzing a random neighboring solution decreases significantly. This reduces the effectiveness of the algorithm for multidimensional problems.

Simulated Annealing is a stochastic algorithm because a different path is created each time the algorithm is run. This is a basic **metaheuristic method**.

7.23 Introduction to population methods and genetic algorithms

The *simulated annealing* algorithm is a good introduction to population algorithms. Their purpose is to increase the effectiveness of the algorithms by performing several runs of the algorithms with different starting parameters. A larger number of runs in the case of algorithms containing randomness increases the chance of obtaining the correct result. Additionally, individual runs of the algorithms can share some information, which will further increase their chances of finding the extreme of the function.

Population methods in optimization problems is an approach in which solutions are represented as individuals in a population. These individuals are subject to a process of improvement, often through genetic processes inspired by biological evolution. The primary goal of these algorithms is to find the optimal solution in the search space by exploring and exploiting potential areas.

The most popular population methods include:

- Genetic algorithms that are based on the mechanisms of genetic inheritance, mutation and crossovers,
- Evolutionary strategies, which focus on evolving solutions by modifying their strategies rather than specific genotypes,
- Swarm algorithms (Particle Swarm Optimization), which are based on the swarm model, where solutions are represented by particles, and the optimization process involves adjusting the movement of particles in the search space based on their experiences.

All these methods have common inspiration from evolutionary processes, and their effectiveness depends on appropriate adjustment of parameters and problem representation. Population algorithms are often used to solve optimization problems in various fields, such as engineering, life sciences, finance, and artificial intelligence.

131890-S Optimization methods **Winter 2024/2025**

Lecture 8: Introduction to discrete optimization

Daniel Kaszyński

8.24 Introduction to discrete optimization

Discrete optimization is a field that deals with problems where at least one of the variables takes on discrete values. These problems are ubiquitous and can be encountered in various areas, including:

1. **Logistics** - managing the time needed for unloading, loading, verification, and transportation of goods, as well as route planning,
2. **Energy management** - adjusting energy production based on demand,
3. **Scheduling** - planning schedules for students and lecturers, determining who, when, with whom, and where.
4. **Sports game schedules** - planning game schedules, including who plays, when, at which stadium, considering audience attendance and broadcast time to maximize profits.

Discrete optimization problems are typically NP problems (nondeterministic polynomial). These are the problems for which the time required to find the optimal solution increases exponentially with the number of elements. Thus, we can quickly find solutions only for small problems, but these are often too small to meet our needs in the real world.

This issue can be illustrated with a graph shown in Figure 8.31. The X-axis represents the number of elements for a given problem, and the Y-axis represents the time needed to find the optimal solution. In an ideal world, we would like to have an algorithm that solves the problem in a linear time. However, due to the complex nature of the problem, the number of possible solutions grows nonlinearly.

In the NP-type problems, we can often determine at a glance whether a proposed solution is valid, but we cannot easily ascertain if it is an optimal one due to the vast number of potential solutions. Given the complexity of discrete problems, two main approaches are considered for solving them:

1. **Shifting the exponential growth line** - adjusting the growth of the time needed so that we can solve larger problems before the solution time becomes a significant issue,
2. **Finding approximate solutions** - seeking solutions that are not optimal but provide satisfactory results in a much shorter time.

To further explore discrete optimization problems, let's take a closer look at specific optimization problems, examining their complexity and possible approaches for solving them.

8.25 The knapsack problem

Imagine a situation where a thief breaks into an art museum intending to steal art and sell them for profit. The thief brings a backpack, but it has a limited carrying capacity (ignore their dimensions for simplicity),

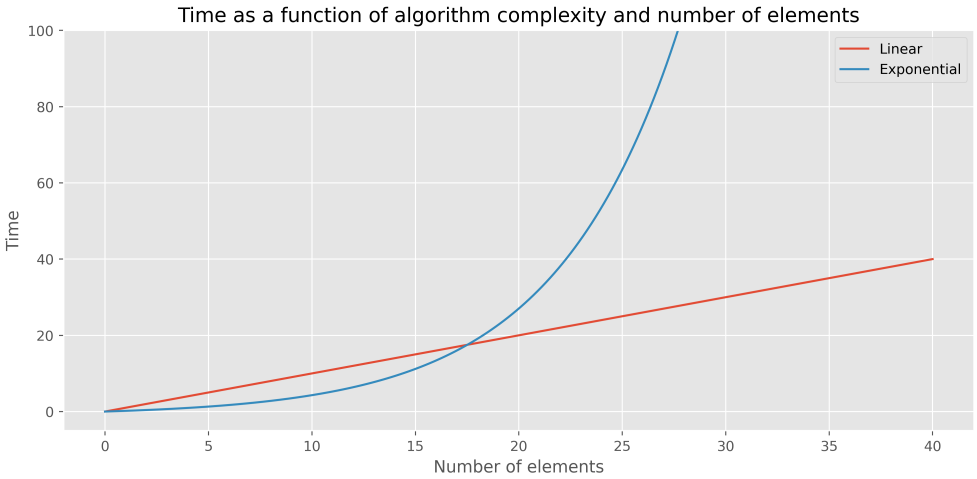


Figure 8.31: Time required to solve a problem depending on the number of elements and computational complexity

which means the thief cannot take all the artworks. Therefore, the thief must decide which items to choose in order to take the most valuable artworks that will fit in the backpack and yield the highest profit.

Let's formalize the problem mathematicaly: we have a set of all available items in the museum $\mathcal{I} = (i_1, i_2, \ldots, i_n)$. Each item has its own value and weight $i_i = (v_i, w_i)$. We also have the maximum carrying capacity of the backpack K and an additional decision variable $x = (x_1, x_2, \ldots, x_n)$, where $x_i = 1$ if the item is placed in the backpack and $x_i = 0$ otherwise. We can now describe the problem as an optimization task aiming to maximize the value of the backpack considering its backpack capacity as a constrain:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{N} w_i x_i \leq K \\ & x_i \in \{0,1\}, \quad \forall i \in \{1,2,\ldots,N\} \end{aligned} \tag{8.62}$$

This is a classic example of the **Knapsack Problem**, a fundamental problem in combinatorial optimization. The goal is to choose the most valuable combination of items that fit within the given weight limit.

Before we move on to discussing approaches to solving this problem, it is important to consider how long it would take to check all possible solutions, in other words conducting a thorough search. The decision variable x can take the following values: $(0, 0, \ldots, 0)$, $(0, 0, \ldots, 1)$, $\ldots$, $(1, 1, \ldots, 1)$. The number of all possible combinations is related to the number of elements in the set $\mathcal{I}$, and in this case, it is equal to $2^{|\mathcal{I}|}$. Assuming it takes one millisecond to check a single case, for a 50-element set $|\mathcal{I}| = 50$, it would take 35677 years to check all possible cases. Therefore, we can see that such an approach must be ruled out as it is practically infeasible.

In the following sections, we will consider approaches that utilize methods such as **greedy search**, **dynamic programming**, and **branch and bound** to design dedicated optimization algorithms - heuristics that find feasible solutions in significantly shorter times.

8.26 Greedy search

Greedy search algorithms are a good strategy for finding an initial correct solution that meets the conditions of an optimization problem. These algorithms conceptually build a solution to the problem in a very simple and intuitive way. Generally, such algorithms may not be the most efficient, but they serve as a first step toward a deeper understanding of the problem, which can later be useful when applying more sophisticated optimization methods. Let's examine the problem described by an equation (8.63):

$$\begin{aligned} & x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \\ & 1x_1 + 1x_2 + 1x_3 + 10x_4 + 11x_5 + 13x_6 + 7x_7 \\ & 2x_1 + 2x_2 + 2x_3 + 5x_4 + 5x_5 + 8x_6 + 3x_7 \leq 10 \end{aligned} \tag{8.63}$$

We have 7 items at our disposal, first equation describes item assignment, second is the objective function with assigned values of the items in dollars (1 dollar, 1 dollar, ...), while the inequality contains the weights of these items in kg (2kg, 2kg, ...) along with the maximum load capacity of the backpack $K = 10$ kg. A potential strategy for solving the problem could be to initially choose the heaviest items and then fill the backpack so as not to exceed the maximum load:

$$\begin{aligned} & x = (1, 0, 0, 0, 0, 1, 0) \\ & 1 * 1 + 1 * 0 + 1 * 0 + 10 * 0 + 10 * 0 + 13 * 1 + 7 * 0 = 14 \\ & 2 * 1 + 2 * 0 + 2 * 0 + 5 * 0 + 5 * 0 + 8 * 1 + 3 * 0 = 10 \leq 10 \end{aligned} \tag{8.64}$$

Let's try a few other strategies. Let's go with "the more, the better" approach by filling the backpack with the lightest items first:

$$\begin{aligned} & x = (1, 1, 1, 0, 0, 0, 1) \\ & 1 * 1 + 1 * 1 + 1 * 1 + 10 * 0 + 10 * 0 + 13 * 0 + 7 * 1 = 10 \\ & 2 * 1 + 2 * 1 + 2 * 1 + 5 * 0 + 5 * 0 + 8 * 0 + 3 * 1 = 9 \leq 10 \end{aligned} \tag{8.65}$$

Changing the strategy resulted in not fully utilizing the available space in the backpack, and its value decreased by 4 dollars.

The next strategy we will take requires additional calculations: we choose items with the highest value per kg of weight, which means for item x_1 we have $\frac{value}{weight} = \frac{1}{2}$. Let's create an temporary list with the items Cost Efficiency, which we will use to guide item selection:

$$CE = \left[\frac{1}{2},\ \frac{1}{2},\ \frac{1}{2},\ 2,\ 2,\ 1\frac{5}{8},\ 2\frac{1}{3}\right] \tag{8.66}$$

Therefore, by selecting items based on their highest values from the list CE, we will choose sequentially $(x_7, x_4, x_5, x_6, x_1, x_2, x_3)$. However, we must consider the backpack's maximum weight capacity, as choosing (x_7, x_4, x_5) will exceed the 10 kg limit:

$$\begin{aligned} & x = (1, 0, 0, 1, 0, 0, 1) \\ & 1 * 1 + 1 * 0 + 1 * 0 + 10 * 1 + 10 * 0 + 13 * 0 + 7 * 1 = 18 \\ & 2 * 1 + 2 * 0 + 2 * 0 + 5 * 1 + 5 * 0 + 8 * 0 + 3 * 1 = 10 \leq 10 \end{aligned} \tag{8.67}$$

We managed to develop an even better strategy, but the question remains whether this is the best strategy. It is easy to see that by selecting items x_4 and x_5, the backpack's value is 20 dollars. How can we develop an algorithm that allows us to achieve this result. But we should also ask ourselves: is this really the optimal solution?

Let's summarize greedy algorithms. We know that we can propose many different algorithms to solve a given problem, some being more or less efficient than others. Greedy algorithms are simple in concept and implementation and generally operate very quickly. However, they have a limitation - they do not always find the optimal solution, and their effectiveness can vary based on the input data. Despite these limitations, greedy algorithms are still useful for creating benchmarks and comparing the performance of other optimization methods for a given problem.

8.26.1 Dynamic programming

Dynamic programming is a problem-solving technique that utilizes the approaches of *divide and conquer* and *bottom-up computation*. In the divide and conquer approach, we break down the problem into smaller subproblems and use them to find the optimal solution. Let's denote the optimal solution by $O(k, j)$, where $k \in [0, K]$ represents the optimal capacity of the backpack, and $j \in [0, n]$ represents the optimal number of items in the backpack. In this approach, instead of solving the problem for all n items at once, we solve it for each successive j.

Let us assume we know how to solve $O(k, j-1)$ for all $k \in [0, K]$, and we want to find out how to solve $O(k, j)$, which means adding item j to the backpack:

1. If the weight of the next item w_j is greater than the k-th maximum load of the backpack then the best solution is $O(k, j-1)$. This is because the item j does not fit given current capacity.
2. In the other case when $w_j \leq k$, we can consider two scenarios and select one that gives a better outcome: (1) do not select the j-th item the best solution is $O(k, j-1)$. (2) select the j-th item the solution is the value of an item j and optimal value of previously considered backpack with reduced capacity by item j: $v_j + O(k - w_j, j-1)$.

Let us write this algorithm in the form of an equation (8.68):

$$O(k,j) = \begin{cases} max(O(k,j-1), v_j + O(k-w_j, j-1)) & \text{if } w_j \leq k \\ O(k,j-1) & \text{else} \end{cases} \tag{8.68}$$

As we can see, it is a recursive equation that can be easily implemented in R, as shown in the code listing 29. Note that the recursive problems require base case, which allows us to stop the computation if we cannot reduce the problem any further. The base case here is specified for *item_idx==0*, meaning that we don't add any items to the backpack (it has no value).

```
N <- 3  # maximum number of items
K <- 9  # maximum backpack capacity
# value and weight of following items
item_values <- c(5, 6, 3)
item_weights <- c(4, 5, 2)

O_ALG <- function(capacity, item_idx) {
  # 0 as no item index -> no value
  if (item_idx == 0) {
    return(0)
  }
```

```
12
13    # select currently asked for item
14    value <- item_values[item_idx]
15    weight <- item_weights[item_idx]
16
17    # can we add the item to the backpack?
18    if (weight <= capacity) {
19      # select the maximum of
20      # 1. case where we don't add the item and consider another item
21      # 2. case where we add the item and consider another item
22      return(max(
23        O_ALG(capacity, item_idx - 1),
24        value + O_ALG(capacity - weight, item_idx - 1)
25      ))
26    }
27    # if not then check the next item
28    return(O_ALG(capacity, item_idx - 1))
29  }
30
31  cat(sprintf("Optymalna wartosc plecaka: %d\n", O_ALG(K, N)))
```

Listing 29: Knapsack problem recursive approach

Adopting such a recursive algorithm is computationaly inefficient approach. Those familiar with the recursive implementation of finding the n-th element of the Fibonacci sequence know that the problem lies in repeated calculating of the same values. This method is known as the *top to bottom* approach. To improve the efficiency of the algorithm, we need to change the direction of execution and adopt the *bottom-up* approach. Instead of starting from the j-th element downwards, we will now start from the zeroth element (no items in backpack) and increase by one until we reach the N-th element. Let's consider this with a simple example:

$$\begin{aligned} \max \quad & 5x_1 + 6x_2 + 3x_3 \\ \text{s.t.} \quad & 4x_1 + 5x_2 + 2x_3 \leq 9 \end{aligned} \tag{8.69}$$

In dynamic programming we use a table that contains information about the optimal values of the backpack for all possible configurations of k and j. This table has rows for $K \in [0, K]$ capacities and columns for $j \in [0, N]$ items. The process of building subsequent columns of the table is shown in Figure 8.32. The first column is initially filled with zeros because if no items are placed in the backpack, its value is zero.

In the second column, we consider placing the first item in the backpack. Using the decision rule describe by equation (8.68), we know that for $k < 4$ we take the values from the previous column as we have no room in smaller backpack. For $k \geq 4$, we consider the equations in the form of $max(O(k, 0), 5 + O(k - 4, 0))$. In this case, the situation is simple because for each k, we take $5 + O(k - 4, 0)$, so filling in this column is straightforward. Note that we no longer need to compute $O(k, 0)$ and $O(k - 4, 0)$ as we have their values stored in a table.

Interesting things occur when we start filling the third column. Since the second item has a weight of $w_2 = 5$, for a backpack with a capacity of $k = 4$, the effective backpack is $O(4, 1)$. But for $k = 5$, we consider $max(O(5, 1), 6 + O(5 - 5, 1)) = max(5, 6) = 6$. Furthermore, for $k = 9$, the equation becomes $max(O(9, 1), 6 + O(9 - 5, 1)) = max(5, 6 + 5) = 11$, which is currently the highest backpack value.

We need to repeat the procedure one more time for the last column to gather all the necessary information. The maximum value we can achieve in the backpack is 11. Interestingly, the highest value will always be in the bottom right corner of the table.

The remaining question is which items should place in the backpack to achieve this value. In this case, we know that it is i_1 and i_2 because we observed this while manually building the table. However, we can easily verify this after building the table without storing all the information along the way.

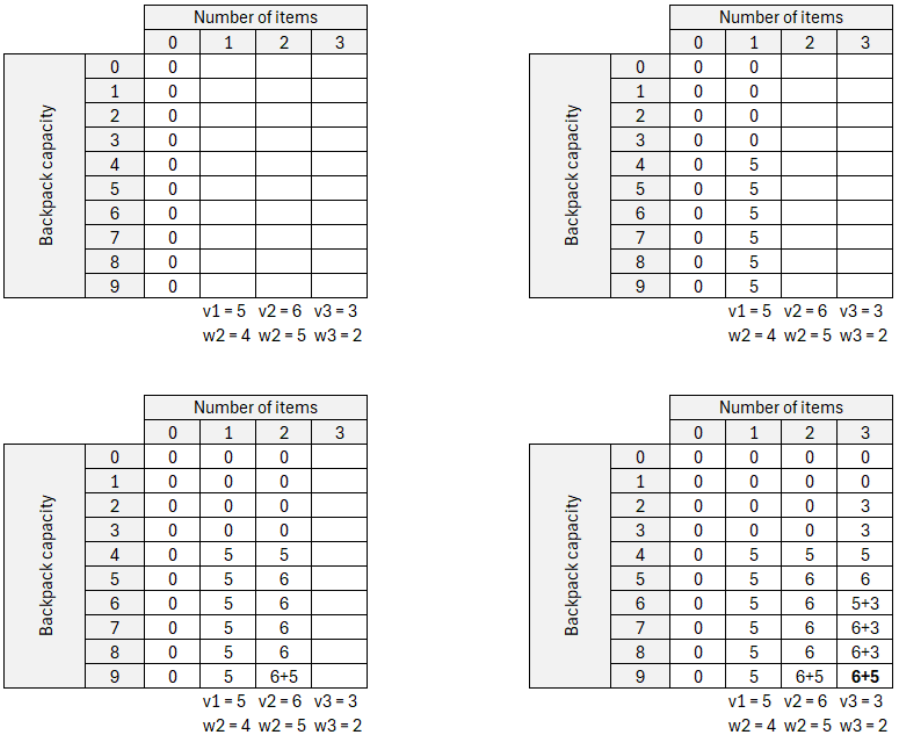


Figure 8.32: Dynamic programming table

Let's take a closer look at Figure 8.69, which shows the complete table with summed up values. To determine which items to place in the backpack, we need to trace back from the optimal basket to the empty one. Starting from position $O(9, 3)$, we check the position to the left, $O(9, 2)$. If the value has not changed, it means that we did not place item j in the backpack. Removing it from consideration does not change the optimal value because it was not in the backpack so item i_3 is not in the optimal backpack. Next, we compare position $O(9, 2)$ with $O(9, 1)$, in this case, the value changed, indicating that removing item i_2 affects the optimal backpack value, so we know it must be in the backpack. In the next step, we do not start at $O(9, 1)$ but at $O(9 - 5, 1)$, which accounts for weight of item i_2. Comparing $O(4, 1)$ with $O(4, 0)$ tells us that item i_1 is placed in the backpack, achieving the optimal value. Thus, the optimal backpack contains items i_1 and i_2.

Code listing 30 presents the implementation of the dynamic programming table algorithm for the knapsack problem and finding the optimal value and contents of the backpack. It is important to note that this algorithm is not without its drawbacks. Compared to the recursive implementation, it reserves memory space for the table with all possible solutions, which can be a significant problem with a large number of items and a high maximum backpack weight.

```
N <- 3  # maximum number of items
K <- 9  # maximum backpack capacity
# value and weight of following items
item_values <- c(5, 6, 3)
item_weights <- c(4, 5, 2)


get_dp_table <- function() {
  # Builds a DP table and fills the entries
```

Backpack capacity	Number of items 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 table - traceback

```
  dp_table <- matrix(0, nrow = K + 1, ncol = N + 1)

  # calculating the DP table
  for (item in 1:N) {
    value <- item_values[item]
    weight <- item_weights[item]
    # starting from the first item, can we add it?
    for (capacity in 0:K) {
      # check this condition for each possible capacity configuration
      if (weight <= capacity) {
        # if its possible to add it then get best possible case out of
        # 1. not adding the item
        # 2. adding the item
        dp_table[capacity + 1, item + 1] <- max(
          dp_table[capacity + 1, item],
          value + dp_table[capacity - weight + 1, item]
        )
      } else {
        # if its not possible to add the item use last item best value function
        dp_table[capacity + 1, item + 1] <- dp_table[capacity + 1, item]
      }
    }
  }

  return(dp_table)
}

# obtaining the optimal solution item configuration
solution_values <- function(dp_table) {
  # Extracting information about optimal solution from the 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
    if (dp_table[current_capacity + 1, current_item + 1] != dp_table[current_capacity + 1,
    current_item]) {
      # since there is a difference that means we should add the item to the backpack

      # save its index and value, weight contribution
      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

      # reduce currently allowed capacity
      current_capacity <- current_capacity - weight
    }
    # move to the next column where we consider adding next item
    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 is a method for solving optimization problems by constructing a decision tree. An example of decision tree for a knapsack with three items is shown in Figure 8.34. The state at the very top of the tree, called the root and represents an empty backpack. The left branch of the tree considers scenarios with the first item in the backpack, and the right branch without it. For these two nodes, we then consider adding the second item to the backpack resulting in four new branches. Further, we consider adding the third item, creating a total of $2^4 - 1 = 15$ states representing all combinations along with intermediate states.

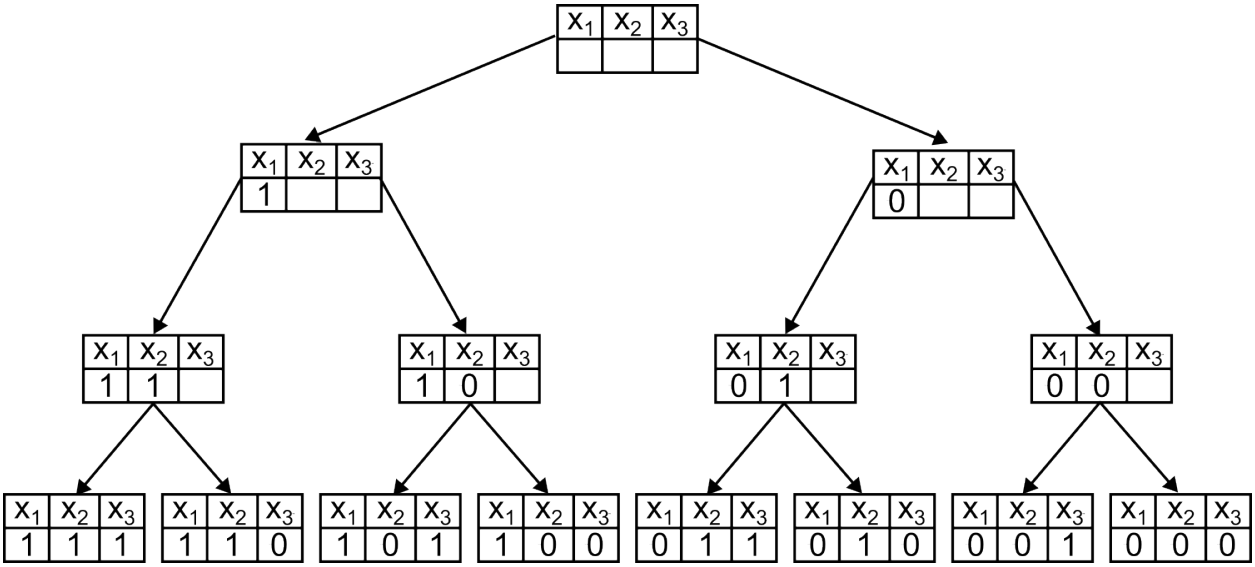


Figure 8.34: Decision tree for the knapsack problem

When the number of items $|\mathcal{I}|$ is too large, this approach becomes impractical, as it would be impossible to check all possible scenarios. Therefore, we need to find a method for generating trees in a way that does not search the entire tree, but make decisions at individual nodes to determine whether it is worthy further exploration. Let's break this problem down into two stages:

1. **branching** - expanding the node of a tree into two nodes,
2. **bounding** - optimistic evaluation by *relaxation* the constraints of the optimization problem.

To understand the concepts of optimistic evaluation and relaxation, let's consider a simple example. We have three items $i_1 = (v = 45, w = 5)$, $i_2 = (v = 48, w = 8)$, and $i_3 = (v = 35, w = 3)$ with $K = 10$, meaning we can never fit all items into the backpack, but we can take only i_2 or i_1 and i_3. By removing the constraint of the maximal load, our optimistic evaluation is the value $45 + 48 + 35 = 128$. We can now start searching the tree while keeping the future optimistic value in mind for each node, comparing and discarding unprofitable branches.

Let's consider the logic of such an algorithm with an example. Figure 8.35 shows the first two stages of tree branching. At the root, we determine the current states, i.e., the backpack value as 0 (no iteams), the remaining free space and the optimistic backpack value at current state. We expand the root into two nodes: in one, item i_1 goes into the backpack, and in the other, we reject it and update the node states. The optimistic value of the nodes differs because right nodes $x = (0, ?, ?)$, no longer include i_1 in the optimistic evaluation so $48 + 35 = 83$ at best.

Let's further expand node $x = (1, ?, ?)$. We can see that in node $x = (1, 1, ?)$, we exceed the maximum load of the backpack, so we can immediately discard further exploration of this path as it will never yield a correct solution. In the case of $x = (1, 0, ?)$, the expected value decreases to 80. Further actions may differ depending on the adopted tree search algorithm, but for now, let's continue exploring node $x = (1, 0, ?)$.

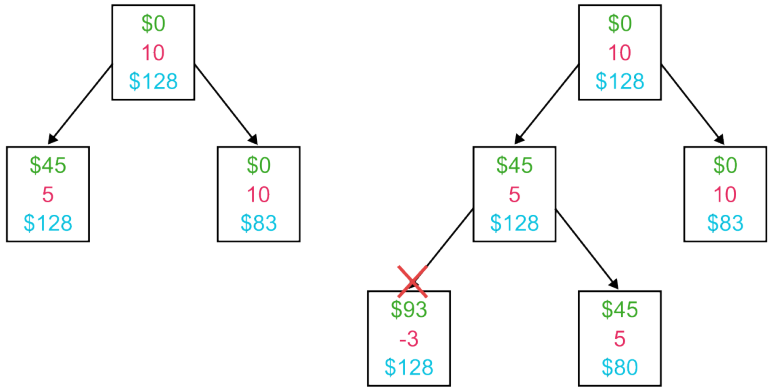


Figure 8.35: Tree branching 1

Further exploration of the tree is shown in Figure 8.36. After expanding node $x = (1, 0, ?)$, we obtain case $x = (1, 0, 1)$, where the backpack value is the highest and equal to the optimistic value of 80. But we also get a slightly worse solution, $x = (1, 0, 0)$ with an optimal value 45.

Since in the relaxation approach we track the highest optimistic value, while browsing the tree, we see that node $x = (0, ?, ?)$ still potential offers a higher optimistic value than the current best solution (80 vs 83). As this could potentially be a better solution, the algorithm continues to search the tree. We can discard node $x = (0, 0, ?)$ because, compared to the optimal solution, it is worse and would not yield a better result. Expanding node $x = (0, 1, ?)$ reveals another optimal solution, $x = (0, 1, 0)$, which is worse than the current best, and an infeasible solution $x = (0, 1, 1)$ because it exceeds the backpack capacity.

Thus, we can see that the use of relaxation and optimistic value allowed us to reduce the size of the tree. However, this result may be unsatisfactory, as we only reduced the tree by two nodes. Therefore, we could find a better relaxation method, which can be crucial for the problems with large trees.

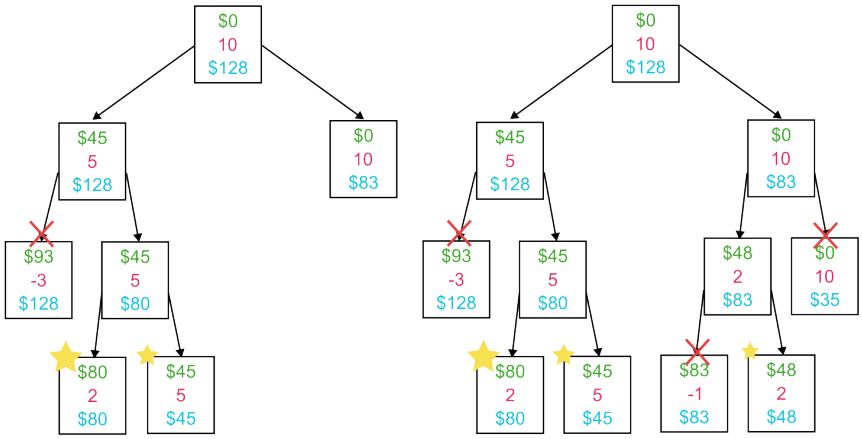


Figure 8.36: Rozwijanie drzewa 2

An interesting example of such a rule might be breaking the discrete nature of the problem by allowing fractional parts of items to be placed in the backpack. We can add the concept from the previously presented greedy algorithm to sort items by their $\frac{\text{value}}{\text{weight}}$, and in the optimistic value, we fill the backpack with a fraction of the item exceeding the load. In our example, this will change the initial optimistic value to:

$$\begin{aligned} CE &= \left[\frac{45}{5}, \frac{48}{8}, \frac{35}{3}\right] = \left[9,\ 6,\ 11\frac{2}{3}\right] \\ w_3 + w_1 + \frac{1}{4}w_2 &= 10 \leq 10 \\ OptimisticValue &= v_3 + v_1 + \frac{1}{4}v_2 = 35 + 45 + 12 = 92 \end{aligned} \tag{8.70}$$

In other words, we reparameterize the problem by introducing $x_i = \frac{y_i}{v_i}$, which results in the following change to the equations of the optimization problem:

$$\begin{aligned} \max \quad & \sum_{\mathcal{I}} y_i \\ \text{s.t.} \quad & \sum_{\mathcal{I}} \frac{w_i}{v_i} y_i \leq K \\ & y_i \in [0, 1] \end{aligned} \tag{8.71}$$

Such an approach means that after expanding the entire node $x = (1, ?, ?)$ and returning to $x = (0, ?, ?)$, we will not continue expanding the tree because the optimistic value after excluding i_1 is $35 + \frac{7}{8} \cdot 48 = 77$ ($\frac{7}{8}$ because we first place item i_3 with a weight of 3 and fill the rest with i_2 with a weight 8).

Beyond the importance of the relaxation method in tree search, we can also discuss approaches related to tree traversal. Popular approaches include **Depth-first search** and **Best-first search**. The depth-first search approach is the method we have discussed in the earlier example. In this approach, we first focus on the left (or right) branches of the tree. Once we reach the end of a branch, we check the remaining branches left behind. In the best-case scenario, we can complete the search after expanding only one branch.

Best-first search works quite differently. In this algorithm, we expand the nodes with the currently highest optimistic value. However, there is no simple answer as to which approach is better, and the time required to solve the problem can varies based on the problem. Best-first search carries additional risk in situations where the values of several items are infinitely large. In such cases, we might skip from node to node, almost recreating the entire tree, while in the depth-first approach, we could terminate the search after traversing one or more branches.

Listing 31 presents the implementation of the branch and bound algorithm using Depth-First search with a simple relaxation that removes the constraint of the backpack's maximum load.

```
N <- 3  # maximum number of items
K <- 9  # maximum backpack capacity
# value and weight of following items
item_values <- c(45, 48, 35)
item_weights <- c(5, 8, 3)

Node <- function(level, value, capacity) {
  return(list(
    level = level,
    value = value,
    capacity = capacity,
    items = c(),
    opt_estimate = 0
  ))
}

optimistic_value_estimation <- function(node) {
  if (node$capacity > K) {
    return(0)
  }

  estimate <- node$value
  new_level <- node$level + 1

  for (i in new_level:N) {
    estimate <- estimate + item_values[i]
  }

  return(estimate)
}

    DFS <- function() {
  max_profit <- 0  # final backpack value
  not_visited <- list()  # stack of nodes
  branching_count <- 0  # count of branched nodes
  opt_items <- c()  # optimal items solution

  # defining root node
  root <- Node(0, 0, 0)
  root$opt_estimate <- optimistic_value_estimation(root)
  not_visited <- append(not_visited, list(root),1)

  while (length(not_visited) > 0) {
    # extracting node from the stack
    parent <- not_visited[[1]]
    not_visited[[1]] <- NULL
    cat(parent$level, parent$value, "\n")
    branching_count <- branching_count + 1  # count number extended branches
    # is the optimistic value of the parent node better than the current best node
    if (parent$opt_estimate > max_profit) {
      ### BRANCH LEFT, we consider adding item to the backpack
      child <- Node(
        parent$level + 1,  # increase node level
```

```
        parent$value + item_values[parent$level + 1],  # value increases
        parent$capacity + item_weights[parent$level + 1]  # weight increases
      )
      child$items <- c(parent$items, parent$level + 1)
      child$opt_estimate <- optimistic_value_estimation(child)

      # saving node optimistic value is better than current profit and doesn't exceed backpack capacity
      if (child$capacity <= K && child$value > max_profit) {
        print(child$capacity)
        max_profit <- child$value
        opt_items <- child$items
      }
      # if the child optimistic value is greater we can branch it (DFS ADD ITEM AT THE BEGINNING OF THE STACK)
      if (child$opt_estimate > max_profit && child$level < N) {
        not_visited <- append(list(child), not_visited, 1)
      }

      ### BRANCH RIGHT, don't add the item to the backpack
      child2 <- Node(
        parent$level + 1,
        parent$value,
        parent$capacity
      )
      child2$items <- parent$items
      child2$opt_estimate <- optimistic_value_estimation(child2)

      # saving node optimistic value is better than current profit and doesn't exceed backpack capacity
      if (child2$capacity <= K && child2$value > max_profit) {
        max_profit <- child2$value
        opt_items <- child2$items
      }
      # if the child optimistic value is greater we can branch it (DFS ADD, but allow searching current branch first)
      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: Knapsack branch and bound algorithm