AI505/AI801, Optimization – Exercise Sheet 02

2026-03-04

Solutions included.

Exercises with the symbol $^+$ are to be done at home before the class. Exercises with the symbol * will be tackled in class. The remaining exercises are left for self training after the exercise class. Some exercises are from the text book and the number is reported. They have the solution at the end of the book.

Exercises

Exercise 1 * (3.6)

Suppose we have a unimodal function defined on the interval $[1, 32]$. After three function evaluations of our choice, will we be able to narrow the optimum to an interval of at most length 10? Why or why not? How much more can we reduce with one further evaluation?

The best we can do is to use Fibonacci Search. With 3 evaluations the uncertainty is shrunk by a factor of $F_{n+1} = F_4 = 3$; that is, to $(32-1)/3 = 10 + 1/3 > 10$. With 4 evaluations it is shrunk by a factor of $F_{n+1} = F_5 = 5$; that is to $(32-1)/5 = 6.2$. So 4 would be necessary to reduce the interval to at most length 10. The uncertainty shrinks by a factor of $10.3/6.2 = 1.66$ by adding one more evaluation after three or: $\frac{F_{n+1}}{F_n} = \frac{F_5}{F_4} = 1.66$ (quite close to the golden ratio 1.61803 of the Golden ratio search).
Some more details on the Fibonacci search are given below.
Fibonacci search is a *derivative-free*, bracketing method for *unconstrained one-dimensional (univariate) optimization*. It is designed to minimize a *unimodal* function

$$f : [a, b] \to \mathbb{R}$$

when the number of function evaluations is fixed in advance.
Fibonacci search is the best algorithm we can use to reduce the (worst-case) uncertainty of the interval that contains the unique minimizer of unimodular functions.
Setting
Assume:

- \$ f \$ is *unimodal* on $[a, b]$,
- We want to perform *exactly* n function evaluations,
- No derivatives are available.

Let the Fibonacci numbers be defined as:

$$F_0 = 0, \quad F_1 = 1, \quad F_2 = 1, \quad F_k = F_{k-1} + F_{k-2} \quad (k \geq 2).$$

Core Idea

At iteration k, the interval length satisfies:

$$b_{k+1} - a_{k+1} = \frac{F_{n-k+1}}{F_{n-k+2}}(b_k - a_k).$$

The method places test points so that after each elimination step, the remaining interval length is proportional to a Fibonacci number.
This ensures *optimal worst-case interval reduction* when the number of evaluations is fixed.
Algorithm
Let n be the total number of planned evaluations.
Initialization
Set $[a_1, b_1] = [a, b]$. Choose interior points:

$$d_1 = a_1 + \frac{F_{n-1+1}}{F_{n-1+2}}(b_1 - a_1),$$

$$c_1 = a_1 + \left(1 - \frac{F_{n-1+1}}{F_{n-1+2}}\right)(b_1 - a_1).$$

Evaluate:

$$f(c_1), \quad f(d_1).$$

Iterations
For $k = 1, \ldots, n-2$:

- If $f(c_k) \leq f(d_k)$:
  - Discard $[d_k, b_k]$
  - Set $b_{k+1} = d_k$
  - Set $d_{k+1} = c_k$
  - Set

$$c_{k+1} = a_{k+1} + (1 - \frac{F_{n-k}}{F_{n-k+1}})(b_{k+1} - a_{k+1})$$

  - Evaluate $f(c_{k+1})$
- Else ($f(c_k) > f(d_k)$):
  - Discard $[a_k, c_k]$
  - Set $a_{k+1} = c_k$
  - Set $c_{k+1} = d_k$
  - Set

$$d_{k+1} = a_{k+1} + \frac{F_{n-k}}{F_{n-k+1}}(b_{k+1} - a_{k+1})$$

  - Evaluate $f(d_{k+1})$

Only *one new function evaluation* is required per iteration.
Iteration $k = n - 1$

- If $f(c_k) \leq f(d_k)$:
  - Discard $[d_k, b_k]$
  - Set $b_{k+1} = d_k$
  - Set $d_{k+1} = c_k$
  - Set $c_{k+1} = d_{k+1} - \epsilon$ for some small $\epsilon > 0$

- Evaluate $f(c_{k+1})$

- Else ($f(c_k) > f(d_k)$):
  - Discard $[a_k, c_k]$
  - Set $a_{k+1} = c_k$
  - Set $c_{k+1} = d_k$
  - Set $d_{k+1} = c_{k+1} + \epsilon$ for some small $\epsilon > 0$
  - Evaluate $f(d_{k+1})$

Example:
Suppose we have an initial bracket $[a_1, b_1] = [1, 32]$. Let us see how Fibonacci search would behave with only 3 evaluations. The initialization computes two interior points and their evaluations as follows:

$$c_1 = a_1 + \left(1 - \frac{F_3}{F_4}\right)(b_1 - a_1) = 1 + 31\left(1 - \frac{2}{3}\right) = 1 + \left(\frac{1}{3}\right)31 = 1 + 10.33 = 11.33$$
$$d_1 = a_1 + \left(\frac{F_3}{F_4}\right)(b_1 - a_1) = 1 + 31\left(\frac{2}{3}\right) = 1 + 20.67 = 21.67$$

Note that by invoking the Fibonacci recursion $F_k = F_{k-1} + F_{k-2}$, ie, $F_4 - F_3 = F_2$ the above an be rewritten as:

$$c_1 = a_1 + \left(\frac{F_2}{F_4}\right)(b_1 - a_1)$$
$$d_1 = a_1 + \left(\frac{F_3}{F_4}\right)(b_1 - a_1)$$

By this it is easy to see that we have choices for our new intervals, i.e, $[a_1, d_1]$ or $[c_1, b_1]$. Regardless of which interval we choose two conditions are satisfied:

1. The length of these two intervals are equal, i.e., $d_1 - a_1 = b_1 - c_1 = (b_1 - a_1)\left(\frac{F_3}{F_4}\right)$
2. The chosen interval has an interior point where we have already evaluated our function.

Suppose without loss of generality that we choose $[a_1, d_1]$. c_1 is already an interior point and if we were to pick our last point to be $c_1 + \epsilon$, the length of the remaining intervals would be $d_1 - c_1$ and $c_1 + \epsilon - a_1$. As $\epsilon \to 0$, noting that $F_3 - F_2 = F_1$ we can see that these intervals have length: $(\frac{F_1}{F_4})(b_1 - a_1) = \left(\frac{F_2}{F_4}\right)(b_1 - a_1) = \frac{1}{3}(32 - 1) = \frac{31}{3} \approx 10.33$. Using this we can see that 3 evaluations are not sufficient to keep our uncertainty below length 10 and thus we need one more evaluation to achieve this.

Exercise 2 $^+$ (3.1)

Give an example of a problem when Fibonacci search can be applied while the bisection method not.

Fibonacci search is preferred when derivatives are not available.

Exercise 3 $^+$ (3.4)

Suppose we have $f(x) = x^2/2 - x$. Apply the bisection method to find an interval containing the minimizer of f starting with the interval $[0, 1000]$. Execute three steps of the algorithm (you can do this by hand or in Python).

We can use the bisection method to find the roots of $f'(x) = x - 1$. After the first update, we have $[0, 500]$. Then, $[0, 250]$. Finally, $[0, 125]$.

Exercise 4 + (3.5)

Suppose we have a function $f(x) = (x+2)^2$ on the interval $[0, 1]$. Is 2 a valid Lipschitz constant for f on that interval?

> No, the Lipschitz constant must bound the derivative everywhere on the interval, and $f'(1) = 2(1+2) = 6$.

Exercise 5 + (4.2)

The first Wolfe condition requires

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \leq f(\boldsymbol{x}_k) + \beta\alpha\nabla \boldsymbol{d}_k f(\boldsymbol{x}_k)$$

What is the maximum step length α that satisfies this condition, given that $f(\boldsymbol{x}) = 5 + x_1^2 + x_2^2$, $\boldsymbol{x}_k = [-1, -1]$, $\boldsymbol{d}_k = [1, 0]$, and $\beta = 10^{-4}$?

> We have that $\nabla f(\boldsymbol{x}_k) = [2x_{k1}, 2x_{k2}] = [-2, -2]$. Applying the first Wolfe condition to our objective function yields $6 + (-1+\alpha)^2 \leq 7 - 10^{-4} \cdot \alpha \cdot 2$, which can be simplified to $\alpha^2 - 2\alpha + 2 \cdot 10^{-4}\alpha \leq 0$. This equation can be solved to obtain $\alpha \leq 1 - 2 \cdot 10^{-4}$. Thus, the maximum step length is $\alpha = 1.9998$.

Exercise 6 +

Consider the following function $f(x_1, x_2)$:

$$f(x_1, x_2) = x_1^4 - 5x_1^2 + x_2^2$$

We are at $\boldsymbol{x}_0 = [0, 1]$ and we want to move in the descent direction $\boldsymbol{d} = [1, -1]$.

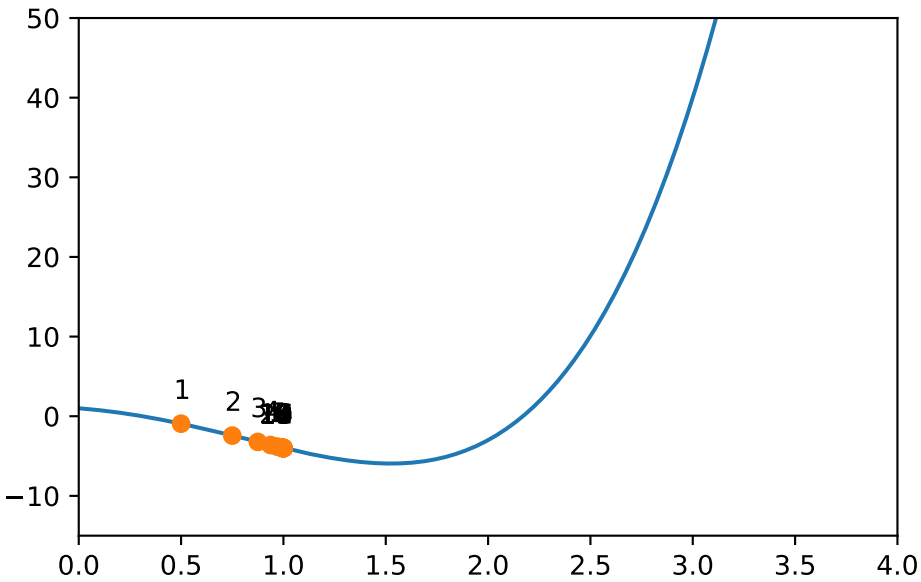
- Plot $f(x_1, x_2)$.
- Define the line search problem to solve.
- Solve the line search problem using the Strong Backtracking Line Search Algorithm to find a value for the learning rate that satisfies the Strong Wolfe conditions.
- Plot the function of the line search and the iterates found in the process.

Note, you find the algorithm in Python in the slides. The focus of this exercise is on visualisation of the search process and on making sense of it.

```python
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

def strong_backtracking(f, nabla, x: np.array, d: np.array, alpha_max: float=1,
    beta: float=1e-4, sigma: float=0.1) -> float:
    y0, g0, y_prev, alpha_prev = f(x), nabla(x) @ d, np.nan, 0
    alpha_lo, alpha_hi = np.nan, np.nan

    alpha=alpha_max
    # bracket phase
    while True:
        alphas = [alpha]
        y = f(x + alpha*d)
        if y > y0 + beta*alpha*g0 or (not math.isnan(y_prev) and y >= y_prev):
            alpha_lo, alpha_hi = alpha_prev, alpha
            break
        g = nabla(x + alpha*d) @ d
        if abs(g) <= -sigma * g0:
            return alphas
        elif g <= 0:
            alpha_lo, alpha_hi = alpha, alpha_prev
            break
        y_prev, alpha_prev, alpha = y, alpha, 2 * alpha

    # zoom phase
    ylo = f(x + alpha_lo*d)
    while abs(alpha_hi - alpha_lo) > 1e-6: # True
        alpha = (alpha_lo + alpha_hi)/2
        alphas.append(alpha)
        y = f(x + alpha * d)
        if y > y0 + beta * alpha * g0 or y >= ylo:
            alpha_hi = alpha
        else:
            g = nabla(x + alpha * d) @ d
            if abs(g) <= -sigma * g0:
                return alphas
            elif g*(alpha_hi -alpha_lo) >= 0:
                alpha_hi = alpha_lo
            alpha_lo = alpha
    return alphas

def plot_the_search(f_alpha, x, d, alphas):
    alpha_samples = np.arange(0, 4., 0.01)
    plt.plot(alpha_samples, f_alpha(alpha_samples,x,d), '-')
    plt.plot(alphas, f_alpha(np.array(alphas),x,d), 'o')
    #plt.text(x, y, '%d, %d' % (int(x), int(y)),
    trans_offset = mtransforms.offset_copy(plt.gca().transData,  fig=plt.gcf(),
                                           x=0.05, y=0.10, units='inches')
    for i in range(len(alphas)):
        plt.text(alphas[i], f_alpha(alphas[i],x,d), '%d' % i,
        transform=trans_offset,
        fontsize=10, verticalalignment='bottom', horizontalalignment='right')
    plt.axis((0, 4, -15, 50))
    plt.show()
```



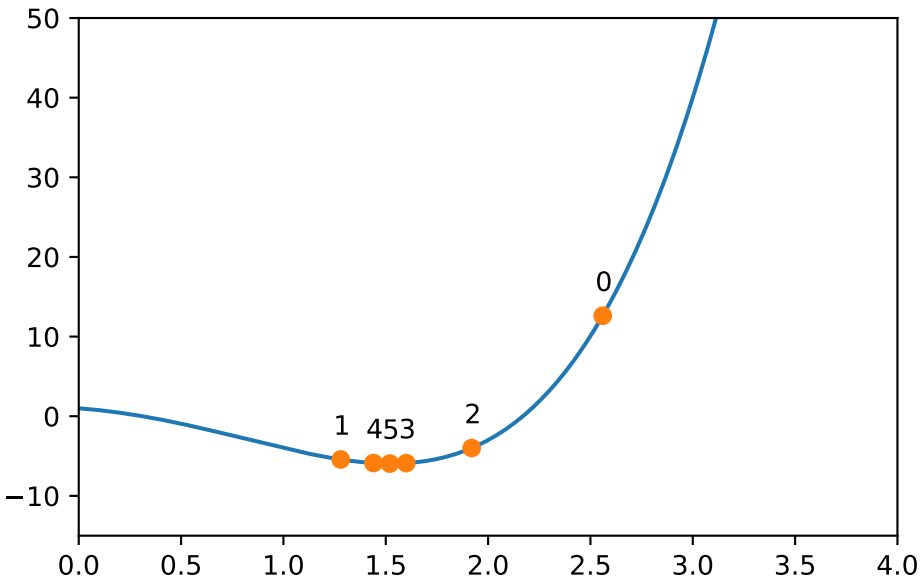
```python
f = lambda x: x[0]**4-5*x[0]**2+x[1]**2
nabla = lambda x: np.array([4*x[0]**3-10*x[0],2*x[1]])

f_alpha = lambda alpha,x,d: (x[0]+alpha*d[0])**4-5*(x[0]+alpha*d[0])**2+\
```



We observe that we do not hit the minimium of the line search function. We can add an initial bracketing phase to find a better α_{max} for the strong backtracking line search. This is done in the code snippet below. We can see that with this addition we are able to find a better step length that satisfies the strong Wolfe conditions and that is closer to the minimum of the line search function.

```
def bracket_minimum(f_alpha, x, d, s=1e-2, k=2.0):
    a, ya = 0, f_alpha(0, x, d)
    b, yb = a + s, f_alpha(s, x, d)
    if yb > ya:
        a, b = b, a
        ya, yb = yb, ya
        s = -s
    while True:
        c, yc = b + s, f_alpha(b + s, x, d)
        if yc > yb:
            return (a, c) if a < c else (c, a)
        a, ya, b, yb = b, yb, c, yc
        s = s*k

alpha_max = bracket_minimum(f_alpha, x, d)
print("alpha_max", alpha_max)
alphas = strong_backtracking(f, nabla, np.array([0,1]), np.array([1,-1]),\
     alpha_max[1])
plot_the_search(f_alpha, x, d, alphas)
```

```
alpha_max (0.64, 2.56)
```



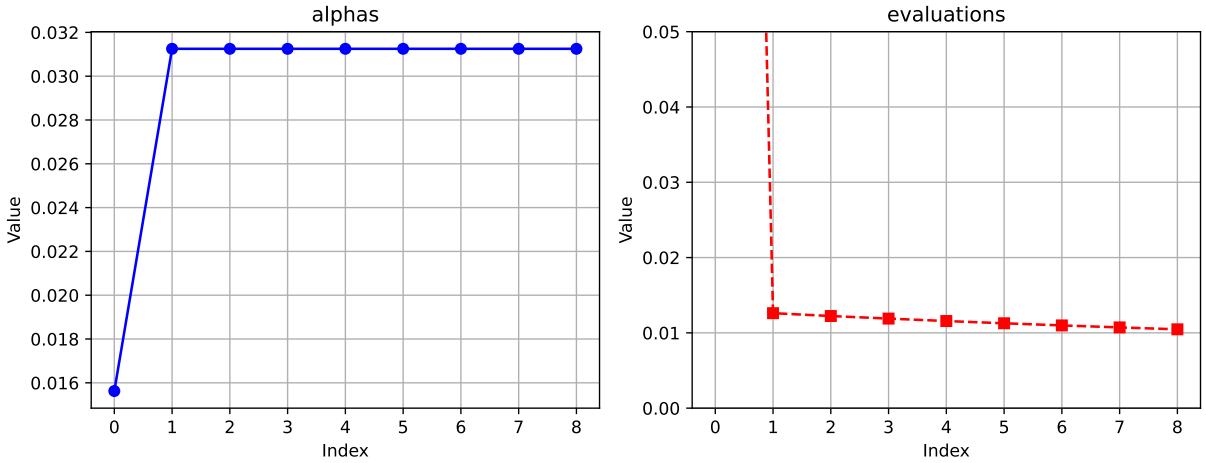
Exercise 7 *

The steepest descent algorithm is a Descent Direction Iteration method that moves along $d_k = -\nabla f(\boldsymbol{x}_k)$ at every step. Program steepest descent algorithms using the backtracking line search. Use them to minimize the Rosenbrock function. Set the initial step length $\alpha_0 = 1$ and print the step length used by each method at each iteration. First, try the initial point $x_0 = [1.2, 1.2]$ and then the more difficult starting point $x_0 = [-1.2, 1]$.

Consider implementing and comparing also other ways for solving the line search problem and the conjugate gradient.

```python
import numpy as np
import matplotlib.pyplot as plt

def rosenbrock(X: np.array, a: int=1, b: int=5):
    """
    Vectorized Rosenbrock function.

    Parameters:
        X: np.ndarray of shape (N, 2) where each row is [x0, x1]
        a: int, default 1
        b: int, default 5

    Returns:
        np.ndarray of shape (N,) with function evaluations.
    """
    X = np.atleast_2d(X)  # Ensure input is always 2D
    x0, x1 = X[:, 0], X[:, 1]  # Extract columns
    return (a-x0)**2 + b*(x1 - x0**2)**2

def gradient(x: np.array, a: int=1, b: int=5):
    # see page 12 of textbook
    return np.array([-2*(a-x[0])-4*b*x[0]*(x[1]-x[0]**2),2*b*(x[1]-x[0]**2)])

def backtracking_line_search(f, grad, x, d, alpha_0=1, p=0.5, beta=1e-4):
    y, g, alpha = f(x), grad(x), alpha_0
    while ( f(x + alpha * d) > y + beta * alpha * np.dot(g, d) ) :
        alpha *= p
    return alpha

def steepest_descent(f, gradient, x_0: np.array, alpha_0: float):
    S=10
    alpha = np.empty((S),dtype=float)
    alpha[0] = alpha_0
    x = np.empty((S,2),dtype=float)
    x[0]=x_0
    last = S-1
    for k in range(S-1):
        alpha[k] = backtracking_line_search(f, gradient, x[k], -gradient(x[k]),alpha_0)
        x[k+1] = x[k] - alpha[k] * gradient(x[k])
        if np.linalg.norm(x[k+1]-x[k], 2) <= 0.001:
            last = k+1
            break
    return x[0:last,:], alpha[0:last], f(x[0:last,:])

points, alphas, evaluations = steepest_descent(rosenbrock, gradient, np.array([1.2,1.2]), 1)

print(points, alphas, evaluations)


# Create the figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(10, 4))  # 1 row, 2 columns

# First subplot
axes[0].plot(alphas, marker='o', linestyle='-', color='b', label='alphas')
axes[0].set_title('alphas')
axes[0].set_xlabel('Index')
axes[0].set_ylabel('Value')
#axes[0].set_ylim(0.028, 0.035)  # Set y-axis limits
axes[0].grid(True)

# Second subplot
axes[1].plot(evaluations, marker='s', linestyle='--', color='r', label='evaluations')
```

```
[[1.2        1.2       ]
 [1.10375    1.2375    ]
 [1.11053542 1.23148877]
 [1.1023775  1.23205131]
 [1.10756431 1.22679658]
 [1.10090929 1.22676599]
 [1.1047616  1.22215202]
 [1.09935593 1.2216352 ]
 [1.102114   1.21755653]] [0.015625 0.03125  0.03125  0.03125  0.03125  0.03125  0.03125  0.03125
 0.03125 ] [0.328      0.01261417 0.01223428 0.0118949  0.01157013 0.01127267
 0.01098867 0.01072334 0.01046936]
```



Note, the exercise is not finished, one should try with different algorithms for solving the line search problem.

Exercise 8 *

Descent direction methods may use search directions other than the steepest descent mentioned in the previous exercise. In general, which descent direction guarantees to produce a decrease in f?

One that makes an angle of strictly less than $\pi/2$ radians with $-\nabla f(\boldsymbol{x}_k)$.
We can verify this claim by using Taylor's theorem updated in directional mode:

$$f(\boldsymbol{x}_k + h\boldsymbol{d}_k) = f(\boldsymbol{x}_k) + h\nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k) + O(h^2) = f(\boldsymbol{x}_k) + h\boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k) + O(h^2).$$

where we used the rule (2.9) in the text book about directional derivative $\nabla_{\boldsymbol{d}} f(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^T \boldsymbol{d} = \boldsymbol{d}^T \nabla f(\boldsymbol{x})$.
When $\boldsymbol{d}_k$ is a downhill direction, the angle θ_k between $\boldsymbol{d}_k$ and $\nabla f(\boldsymbol{x}_k)$ has $\cos\theta_k < 0$, so that

$$\boldsymbol{d}_k^T \nabla f(\boldsymbol{x}_k) = \|\boldsymbol{d}_k\| \, \|\nabla f(\boldsymbol{x}_k)\| \cos\theta_k < 0.$$

It follows that $f(\boldsymbol{x}_k + \boldsymbol{d}_k) < f(\boldsymbol{x}_k)$ for all positive but sufficiently small values of h.

Exercise 9 $^+$ (5.1)

Compute the gradient of $\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$ when A is symmetric. You have the result in the slides of lectures, here you are asked to show that the result is correct.

The real-valued function $f : \mathbb{R}^n \to \mathbb{R}$, $f \overset{def}{=} \boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$ has derivative $\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}x}$ which is a $1 \times n$ matrix, i.e., it is a row vector. The gradient of the function $\nabla f(\boldsymbol{x})$ is its transpose, which is a column vector in $\mathbb{R}^n$ with components the partial derivatives of f:

$$\nabla f(\boldsymbol{x})_i = \frac{\partial f(\boldsymbol{x})}{\partial x_i}, \qquad i = 1, \dots, n$$

A vector-valued function like the linear transformation $g : \mathbb{R}^n \to \mathbb{R}^m$, $g(\boldsymbol{x}) \overset{def}{=} A\boldsymbol{x}$, has derivative $\frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}}$ that denotes the $m \times n$ matrix of first-order partial derivatives of the transformation from x to $g(\boldsymbol{x})$:

$$\frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} \frac{\partial g_1(\boldsymbol{x})}{\partial x_1} & \frac{\partial g_1(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial g_1(\boldsymbol{x})}{\partial x_n} \\ \frac{\partial g_2(\boldsymbol{x})}{\partial x_1} & \frac{\partial g_2(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial g_2(\boldsymbol{x})}{\partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial g_m(\boldsymbol{x})}{\partial x_1} & \frac{\partial g_m(\boldsymbol{x})}{\partial x_2} & \cdots & \frac{\partial g_m(\boldsymbol{x})}{\partial x_n} \end{bmatrix}$$

Such a matrix is called the Jacobian matrix of the transformation $g(\cdot)$.
Since the ith element of g is given by:

$$g_i(\boldsymbol{x}) = \sum_{\ell=1}^{n} a_{i,\ell} x_\ell$$

it follows that

$$\frac{\partial g_i(\boldsymbol{x})}{\partial x_j} = a_{ij}$$

and hence that

$$\frac{\mathrm{d}g(\boldsymbol{x})}{\mathrm{d}\boldsymbol{x}} = A$$

The element-by-element calculations involve cumbersome manipulations and, thus, it is useful to derive the necessary results for matrix differentiation and have them readily available.
Some matrix derivatives useful for us are reported in the table below. They are presented alongside similar-looking scalar derivatives to help memory. This doesn't mean matrix derivatives always look just like scalar ones. In these examples, b is a constant scalar, and B is a constant matrix.

Scalar derivative $f(x) \to \frac{\mathrm{d}f}{\mathrm{d}x}$	Vector derivative $g(\boldsymbol{x}) \to \frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{x}}$
$bx \to b$	$\boldsymbol{x}^T B \to B$
$bx \to b$	$\boldsymbol{x}^T b \to b$
$x^2 \to 2x$	$\boldsymbol{x}^T\boldsymbol{x} \to 2\boldsymbol{x}$
$bx^2 \to 2bx$	$\boldsymbol{x}^T B\boldsymbol{x} \to 2B$

For our specific case we have $\nabla f(\boldsymbol{x}) = 2Ax - b$.
The book "The matrix cookbook" listed among the course references as [MC]. It is a collection of useful formulas for among others matrix differentiation. It is a good reference to have at hand when doing calculations with matrices.

Exercise 10 * (5.2)

Apply one step of gradient descent to $f(x) = x^4$ from $x_0 = 1$ with both a unit step factor and with exact line search.

Given $f(x) = x^4$, we have $f'(x) = 4x^3$ and our starting point is said to be $x_0 = 1$.
One step of gradient descent with unit step works as follows: $x_1 = x_0 - (1)f'(x_0)$ and this gives that $x_1 = 1 - 4 = -3$.
When we do exact line search along the direction of the negative gradient, we are required to solve: $\min_{\alpha \geq 0} f(x_0 - \alpha f'(x_0))$ and in this problem this amounts to solving: $\min_{\alpha \geq 0}(1 - 4\alpha)^4$. The unique minimum is when $\alpha = 1/4$. Now one step using this α value looks like: $x_1 = 1 - 4(1/4) = 0$. Thus exact line search converges to the minimizer in one step starting from $x_0 = 1$.

Exercise 11 +

Recall that a way to measure rate of convergence is by the *limit of the ratio of successive errors*,

$$\lim_{k \to \infty} \frac{f(\boldsymbol{w}_{k+1}) - f(\boldsymbol{w}^*)}{f(\boldsymbol{w}_k) - f(\boldsymbol{w}^*)} = r$$

Different r values of give us different rates of convergence:

- If $r = 1$ we call it a sublinear rate.
- If $r \in (0, 1)$ we call it a linear rate.
- If $r = 0$ we call it a superlinear rate.

Consider the following sequences, which represent the error $e_k = \|F(\boldsymbol{w}_k) - F^*\|$ at iteration k of an optimization algorithm:

1. $e_k = 0.5^k$
2. $e_k = \frac{1}{k+1}$
3. $e_k = 0.1^k$
4. $e_k = \frac{1}{(k+1)^2}$
5. $e_k = \frac{1}{2^{2^k}}$

Tasks:

a) Classify the convergence rate of each sequence as linear, sublinear, superlinear, or quadratic.

b) Provide a justification for each classification by computing the ratio e_{k+1}/e_k or by using the definition of order of convergence.

We analyze each sequence by computing the ratio e_{k+1}/e_k and checking how it behaves as $k \to \infty$.

- $e_k = 0.5^k$

$$\frac{e_{k+1}}{e_k} = \frac{0.5^{k+1}}{0.5} = 0.5$$

Since this ratio is a constant $r = 0.5$ with $0 < r < 1$, the convergence is linear.

- $e_k = \frac{1}{k+1}$

$$\frac{e_{k+1}}{e_k} = \frac{1}{k+2} \Big/ \frac{1}{k+1} = \frac{k+1}{k+2}$$

Taking the limit:

$$\lim_{k\to\infty} \frac{k+1}{k+2} = 1$$

Since the ratio tends to 1, the convergence is sublinear (very slow).

- $e_k = 0.1^k$

$$\frac{e_{k+1}}{e_k} = \frac{0.1^{k+1}}{0.1^k} = 0.1$$

Since this ratio is a constant $r = 0.1$ with $0 < r < 1$, the convergence is linear, but it is faster than Sequence 1 (since $r = 0.1$ is smaller than $r = 0.5$ and closer to 0 which is the superlinear rate).

- $e_k = \frac{1}{(k+1)^2}$

$$\frac{e_{k+1}}{e_k} = \frac{1}{(k+2)^2} \Big/ \frac{1}{(k+1)^2} = \frac{(k+1)^2}{(k+2)^2}$$

Taking the limit:

$$\lim_{k\to\infty} \frac{(k+1)^2}{(k+2)^2} = \left(\frac{k+1}{k+2}\right)^2 = 1$$

This suggests linear convergence.

- $e_k = \frac{1}{2^{2^k}}$

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = \frac{2^{2^k}}{2^{2^{k+1}}} = \frac{2^{2^k}}{2^{2\cdot 2^k}} = \frac{2^{2^k}}{2^2 \cdot 2^{2^k}} = \frac{1}{2^2}$$

This suggests linear convergence. To verify if it is faster than linear, we check:

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^2} = \frac{\left(2^{2^k}\right)^2}{2^{2^{k+1}}} = \frac{\left(2^{2^k}\right)^2}{2^{2\cdot 2^k}} = \frac{\left(2^{2^k}\right)^2}{2^2 \cdot 2^{2^k}} = \frac{2^{2^k}}{2^2} = \infty$$

Since this limit is not finite, the sequence is superlinear but not quadratic.

Exercise 12 *

Consider applying gradient descent to the one-dimensional quadratic function

$$f(x) = \frac{1}{2}x^2$$

with the update rule:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \nabla f(\boldsymbol{x}_k).$$

where $\nabla f(x) = x$.

Tasks:

a) Derive the update formula for $\boldsymbol{x}_k$.

b) Show that the error $e_k = \|\boldsymbol{x}_k\|$ follows an exponential decay when $0 < \alpha < 2$.

c) Compute $\frac{e_{k+1}}{e_k}$ and determine the rate of convergence for different values of α.

d) Set up a Python experiment where gradient descent is applied with different step sizes (α) and verify the theoretical convergence rate numerically.

Hint: Try $\alpha = 0.1, 0.5, 1, 1.5$ and observe how quickly the errors decrease.

The update rule is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \boldsymbol{x}_k = (1-\alpha)\boldsymbol{x}_k$$

Expanding the recursion:

$$\begin{aligned} x_1 &= (1-\alpha)x_0 \\ x_2 &= (1-\alpha)x_1 = (1-\alpha)^2 x_0 \\ x_3 &= (1-\alpha)x_2 = (1-\alpha)^3 x_0. \end{aligned}$$

By induction, the general formula is:

$$x_k = (1-\alpha)^k x_0.$$

The error is defined as $e_k = \|x_k\|$.
Since $x_k = (1-\alpha)^k x_0$, we have:

$$e_k = |(1-\alpha)^k||x_0|$$

Since $0 < \alpha < 2$, the term $|1-\alpha|$ satisfies:

$$|1-\alpha| < 1$$

Thus, as $k \to \infty$, we see exponential decay:

$$e_k = |1-\alpha|^k e_0.$$

c$ This result together with the analysis in the previous exercise confirms that gradient descent converges linearly in this simple quadratic case.
d$

```python
import numpy as np
import matplotlib.pyplot as plt

# Function and gradient
def f(x):
    return 0.5 * x**2

def grad_f(x):
    return x  # Since f(x) = (1/2) x^2, its gradient is simply x

# Gradient Descent Function
def gradient_descent(x0, alpha, num_iters):
    errors = []
    x_k = x0
    for _ in range(num_iters):
        errors.append(abs(x_k))  # Store absolute error |x_k|
        x_k = x_k - alpha * grad_f(x_k)  # Gradient descent update
    return errors

# Initial point
x0 = 10  # Starting point

# Step sizes to test
alphas = [0.1, 0.5, 1, 1.5]

# Number of iterations
num_iters = 50

# Run gradient descent for different values of alpha
plt.figure(figsize=(8, 5))

for alpha in alphas:
    errors = gradient_descent(x0, alpha, num_iters)
    plt.semilogy(errors, label=f"\alpha = {alpha}")

# Plot settings
plt.xlabel("Iteration k")
plt.ylabel("Error |x_k| (log scale)")
plt.title("Gradient Descent Convergence for Different alpha")
plt.legend()
plt.grid()
plt.show()
#plt.savefig("convergence.png")
```

Explanation:

- We initialize $x_0 = 10$ (arbitrary nonzero starting point).
- For each α, we perform gradient descent for 50 iterations.
- We plot e_k on a semilog scale, where exponential decay appears as a straight line.
- Expected behavior:
  - If $0 < \alpha < 2$, the error decreases exponentially.

Higher α (closer to 2) should converge faster but risks instability if $\alpha \geq 2$.

Expected Observations:

- For small α (e.g., 0.1): Convergence is slow.
- For moderate α (e.g., 0.5, 1): Faster convergence.
- For large α (e.g., 1.5): Convergence is still fast but close to instability.
- If we set $\alpha = 2$, we would see oscillations instead of convergence.

The script numerically confirms the exponential decay of errors when $0 < \alpha < 2$.

(5.7)

In conjugate gradient descent, what is the descent direction at the first iteration for the function $f(x, y) = x^2 + xy + y^2 + 5$ when initialized at $[x, y] = [1, 1]$? What is the resulting point after two steps of the conjugate gradient method?

The conjugate gradient method initially follows the steepest descent direction. The gradient is

$$\nabla f(x, y) = [2x + y, 2y + x]$$

which for $(x, y) = (1, 1)$ is $[3, 3]$. The direction of steepest descent is opposite the gradient, $d_0 = [-3, -3]$.
It is possible to prove the following proposition:
Proposition 1. *A twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is* convex*, if and only if the Hessian $\nabla^2 f(x)$ is* positive semi-definite *for all $x \in \mathbb{R}^n$.*
See https://wiki.math.ntnu.no/_media/tma4180/2016v/note2.pdf for a proof.
The Hessian is

$$H = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Since the function is quadratic and the Hessian is positive definite (you can check this by showing that both eigenvalues are positive), the conjugate gradient method converges in at most two steps. Thus, the resulting point after two steps is the optimum, $(x, y) = (0, 0)$, where the gradient is zero.