

# Solutions to Exercises

## Portfolio Optimization: Theory and Application Appendix B - Optimization Algorithms

Daniel P. Palomar (2025). *Portfolio Optimization: Theory and Application*.  
Cambridge University Press.

[portfoliooptimizationbook.com](http://portfoliooptimizationbook.com)

Contributors:

- [Runhao SHI](#)
- [Daniel Palomar](#)

### Exercise B.1: Euclidean norm approximation

- Randomly generate the parameters  $\mathbf{A} \in \mathbb{R}^{10 \times 5}$  and  $\mathbf{b} \in \mathbb{R}^{10}$ .
- Formulate a regression problem to approximate  $\mathbf{Ax} \approx \mathbf{b}$  based on the  $\ell_2$ -norm.
- Solve it directly with the least squares closed-form solution.
- Solve it using a modeling framework (e.g., CVX).
- Solve it invoking a QP solver.

### Solution

- Randomly generate parameters

```
set.seed(123)
A <- matrix(rnorm(10*5), nrow=10, ncol=5)
b <- rnorm(10)
```

- Formulation of the regression problem with l2-norm

$$\min \|\mathbf{Ax} - \mathbf{b}\|_2^2 \quad (1)$$

- Solve with least squares closed-form solution

```
x_ls <- solve(t(A) %*% A) %*% t(A) %*% b
print("Solution via least squares:")
```

```
[1] "Solution via least squares:"
```

```
print(x_ls)
```

```
      [,1]  
[1,]  0.1525699  
[2,] -0.2521882  
[3,] -0.7146094  
[4,] -0.3606949  
[5,]  0.1380150
```

```
print(paste("Objective value:", sum((A %*% x_ls - b)^2)))
```

```
[1] "Objective value: 4.16000670361162"
```

d. Solve using CVX

```
library(CVXR)  
x_var <- Variable(5)  
objective <- Minimize(sum_squares(A %*% x_var - b))  
problem <- Problem(objective)  
result <- solve(problem)  
x_cvx <- result$getValue(x_var)  
print("Solution via CVX:")
```

```
[1] "Solution via CVX:"
```

```
print(x_cvx)
```

```
      [,1]  
[1,]  0.1525699  
[2,] -0.2521882  
[3,] -0.7146094  
[4,] -0.3606949  
[5,]  0.1380150
```

```
print(paste("Objective value:", sum((A %*% x_cvx - b)^2)))
```

```
[1] "Objective value: 4.16000670361162"
```

e. Solve using QP solver

```

library(quadprog)
# For QP solver, we need to rewrite the problem as:
# min (1/2) x^T Q x - c^T x
# Q = 2A^TA, c = 2A^Tb
Q <- 2 * t(A) %*% A
c <- 2 * t(A) %*% b

# The solve.QP function expects parameters in a specific format
# In particular, we need to provide Amat and bvec correctly
# Since we have no inequality constraints, we can use a dummy constraint
result_qp <- solve.QP(
  Dmat = Q,
  dvec = c,
  Amat = matrix(0, nrow = 5, ncol = 1), # Dummy constraint matrix
  bvec = rep(0, 1),                     # Dummy constraint vector
  meq = 0                               # No equality constraints
)

x_qp <- result_qp$solution
print("Solution via QP:")

[1] "Solution via QP:"

print(x_qp)

[1] 0.1525699 -0.2521882 -0.7146094 -0.3606949 0.1380150

print(paste("Objective value:", sum((A %*% x_qp - b)^2)))

[1] "Objective value: 4.16000670361162"

```

### Exercise B.2: Manhattan norm approximation

- Randomly generate the parameters  $\mathbf{A} \in \mathbb{R}^{10 \times 5}$  and  $\mathbf{b} \in \mathbb{R}^{10}$ .
- Formulate a regression problem to approximate  $\mathbf{Ax} \approx \mathbf{b}$  based on the  $\ell_1$ -norm.
- Solve it using a modeling framework (e.g., CVX).
- Rewrite it as an LP and solve it invoking an LP solver.

## Solution

- a. Randomly generate parameters

```
set.seed(123)
A <- matrix(rnorm(10*5), nrow=10, ncol=5)
b <- rnorm(10)
```

- b. Formulation of the regression problem with l1-norm

$$\min \|Ax - b\|_1 \quad (2)$$

- c. Solve using CVX

```
library(CVXR)
x_var <- Variable(5)
objective <- Minimize(cvxr_norm(A %*% x_var - b, 1))
problem <- Problem(objective)
result <- solve(problem)
x_cvx <- result$getValue(x_var)
print("Solution via CVX:")
```

```
[1] "Solution via CVX:"
```

```
print(x_cvx)
```

```
      [,1]
[1,] -0.2129125
[2,] -0.1175085
[3,] -0.1685641
[4,] -0.1271895
[5,] -0.2188999
```

```
print(paste("Objective value:", sum(abs(A %*% x_cvx - b))))
```

```
[1] "Objective value: 5.14315749917209"
```

- d. Rewrite as LP and solve

```
# To rewrite as an LP, we introduce auxiliary variables  $t_i \geq |A_i x - b_i|$ 
# which can be expressed as:
#  $t_i \geq A_i x - b_i$  and  $t_i \geq -(A_i x - b_i)$ 
# Then we minimize  $\sum(t_i)$ 
```

```
library(lpSolve)
```

```
# Set up the LP problem
n <- 5 # dimension of x
m <- 10 # number of rows in A

# Objective function: minimize  $\sum(t)$ 
# decision variables:  $x_{\text{plus}}$  (n),  $x_{\text{minus}}$  (n),  $t$  (m) - total  $2n + m$ 
obj <- c(rep(0, 2*n), rep(1, m))
```

```
# Constraints:  $t_i \geq A_i x - b_i$  and  $t_i \geq -(A_i x - b_i)$ 
# build constraints using ( $x_{\text{plus}} - x_{\text{minus}}$ )
const_mat <- matrix(0, nrow = 2*m, ncol = 2*n + m)
```

```
for (i in 1:m) {
  #  $A_i (x^+ - x^-) - t_i \leq b_i$ 
  const_mat[i, 1:n] <- A[i, ] #  $x^+$ 
  const_mat[i, n + (1:n)] <- -A[i, ] #  $-x^-$ 
  const_mat[i, 2*n + i] <- -1 #  $-t_i$ 

  #  $-A_i (x^+ - x^-) - t_i \leq -b_i$ 
  const_mat[m+i, 1:n] <- -A[i, ]
  const_mat[m+i, n + (1:n)] <- A[i, ]
  const_mat[m+i, 2*n + i] <- -1
}
```

```
const_dir <- rep("<=", 2*m)
const_rhs <- c(b, -b)
```

```
# Solve the LP
lp_result <- lp("min", obj, const_mat, const_dir, const_rhs)

x_recovered <- lp_result$solution[1:n] - lp_result$solution[(n+1):(2*n)]
print("Solution via LP:")
```

```
[1] "Solution via LP:"
```

```
print(x_recovered)
```

```
[1] -0.2129125 -0.1175085 -0.1685641 -0.1271895 -0.2188999
```

```
print(paste("Objective value:", sum(abs(A %*% x_recovered - b))))
```

```
[1] "Objective value: 5.14315749917322"
```

### Exercise B.3: Chebyshev norm approximation

- Randomly generate the parameters  $\mathbf{A} \in \mathbb{R}^{10 \times 5}$  and  $\mathbf{b} \in \mathbb{R}^{10}$ .
- Formulate a regression problem to approximate  $\mathbf{Ax} \approx \mathbf{b}$  based on the  $\ell_\infty$ -norm.
- Solve it using a modeling framework (e.g., CVX).
- Rewrite it as an LP and solve it invoking an LP solver.

### Solution

- Randomly generate parameters

```
set.seed(123)
A <- matrix(rnorm(10*5), nrow=10, ncol=5)
b <- rnorm(10)
```

- Formulation of the regression problem with linf-norm

$$\min \|Ax - b\|_\infty = \min \max_i |A_i x - b_i| \quad (3)$$

- Solve using CVX

```
library(CVXR)
x_var <- Variable(5)
objective <- Minimize(cvxr_norm(A %*% x_var - b, "inf"))
problem <- Problem(objective)
result <- solve(problem)
x_cvx <- result$getValue(x_var)
print("Solution via CVX:")
```

```
[1] "Solution via CVX:"
```

```
print(x_cvx)
```

```
      [,1]
[1,] 0.5999573
```

```
[2,] -0.3496515  
[3,] -0.5369766  
[4,] -0.4804907  
[5,]  0.2606513
```

```
print(paste("Objective value:", max(abs(A %*% x_cvx - b))))
```

```
[1] "Objective value: 0.830211795976401"
```

d. Rewrite as LP and solve

```
# To rewrite as an LP, we introduce a single auxiliary variable t where:
# t >= |A_i x - b_i| for all i
# which can be expressed as:
# t >= A_i x - b_i and t >= -(A_i x - b_i) for all i
# Then we minimize t
```

```
library(lpSolve)
```

```
# Set up the LP problem
```

```
n <- 5 # dimension of x
```

```
m <- 10 # number of rows in A
```

```
# Objective function: minimize t
```

```
# decision vars: x_plus (5), x_minus (5), t (1)
```

```
obj <- c(rep(0, 2*n), 1)
```

```
# Constraints: t >= A_i x - b_i and t >= -(A_i x - b_i)
```

```
# This results in 2m constraints
```

```
const_mat <- matrix(0, nrow = 2*m, ncol = 2*n + 1)
```

```
for (i in 1:m) {
```

```
  # A_i (x^+ - x^-) - t <= b_i
```

```
  const_mat[i, 1:n] <- A[i, ]
```

```
  const_mat[i, n + (1:n)] <- -A[i, ]
```

```
  const_mat[i, 2*n + 1] <- -1
```

```
  # -A_i (x^+ - x^-) - t <= -b_i
```

```
  const_mat[m+i, 1:n] <- -A[i, ]
```

```
  const_mat[m+i, n + (1:n)] <- A[i, ]
```

```
  const_mat[m+i, 2*n + 1] <- -1
```

```
}
```

```
const_dir <- rep("<=", 2*m)
```

```
const_rhs <- c(b, -b)
```

```
# Solve the LP
```

```
lp_result <- lp("min", obj, const_mat, const_dir, const_rhs)
```

```
x_rec <- lp_result$solution[1:n] - lp_result$solution[(n+1):(2*n)]
```

```
print("Solution via LP:")
```

```
[1] "Solution via LP:"
```

```
print(x_rec)
```

```
[1] 0.5999754 -0.3496690 -0.5369699 -0.4804633 0.2606434
```

```
print(paste("Objective value:", max(abs(A %*% x_rec - b))))
```



```
[1] "Objective value: 0.830195475531776"
```

#### Exercise B.4: Solving an LP

Consider the following LP:

$$\begin{array}{ll}\underset{x_1, x_2}{\text{maximize}} & 3x_1 + x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 4, \\ & 4x_1 + 2x_2 \leq 12, \\ & x_1, x_2 \geq 0.\end{array}$$

- Solve it using a modeling framework (e.g., CVX).
- Solve it by directly invoking an LP solver.
- Solve it by invoking a general-purpose nonlinear solver.
- Implement the projected gradient method to solve the problem.
- Implement the constrained Newton's method to solve the problem.
- Implement the log-barrier interior-point method to solve the problem (use (1,1) as the initial point).
- Compare all the solutions and the computation time.

#### Solution

Consider the LP:

$$\underset{x_1, x_2}{\text{maximize}} \quad 3x_1 + x_2 \tag{4}$$

$$\text{subject to} \quad x_1 + 2x_2 \leq 4 \tag{5}$$

$$4x_1 + 2x_2 \leq 12 \tag{6}$$

$$x_1, x_2 \geq 0 \tag{7}$$

- Solve using a modeling framework (CVX):

```

library(CVXR)

# Define variables
x <- Variable(2)

# Define objective and constraints
objective <- Maximize(3*x[1] + x[2])
constraints <- list(
  x[1] + 2*x[2] <= 4,
  4*x[1] + 2*x[2] <= 12,
  x >= 0
)

# Set up and solve problem
prob <- Problem(objective, constraints)
result_cvx <- solve(prob)

# Extract solution
x_cvx <- result_cvx$getValue(x)
obj_val_cvx <- result_cvx$value

# Display results
print("CVX solution:")

[1] "CVX solution:"

print(paste("x1 =", x_cvx[1], "x2 =", x_cvx[2]))

[1] "x1 = 3 x2 = -1.96527429244511e-23"

print(paste("Objective value =", obj_val_cvx))

[1] "Objective value = 9"

```

b. Solve using an LP solver:

```

library(lpSolve)

# Objective coefficients (maximizing 3x1 + x2)
obj <- c(3, 1)

# Constraint matrix
constr <- matrix(c(
  1, 2, # x1 + 2x2 <= 4
  4, 2  # 4x1 + 2x2 <= 12
), nrow = 2, byrow = TRUE)

# Right-hand side of constraints
rhs <- c(4, 12)

# Direction of constraints (all <=)
dir <- c("<=", "<=")

# Solve LP
lp_result <- lp("max", obj, constr, dir, rhs, all.bin = FALSE)

# Extract solution
x_lp <- lp_result$solution
obj_val_lp <- lp_result$objval

# Display results
print("LP solver solution:")

[1] "LP solver solution:"

print(paste("x1 =", x_lp[1], "x2 =", x_lp[2]))

[1] "x1 = 3 x2 = 0"

print(paste("Objective value =", obj_val_lp))

[1] "Objective value = 9"

```

c. Solve using a general-purpose nonlinear solver:

```

library(nloptr)

# Objective function (negative for minimization)
obj_fn <- function(x) {
  return(-(3*x[1] + x[2]))
}

# Constraint function
constraint_fn <- function(x) {
  c(
    x[1] + 2*x[2] - 4,      # <= 0  <=>  x_1 + 2x_2 <= 4
    4*x[1] + 2*x[2] - 12    # <= 0  <=>  4x_1 + 2x_2 <= 12
  )
}

# Set optimization options
opts <- list("algorithm" = "NLOPT_LN_COBYLA",
            "xtol_rel" = 1.0e-9,
            "maxeval" = 1000)

# Run optimization
result_nloptr <- nloptr(
  x0 = c(1, 1),           # Initial point
  eval_f = obj_fn,        # Objective function
  lb = c(0, 0),           # Lower bounds
  ub = c(Inf, Inf),       # Upper bounds
  eval_g_ineq = constraint_fn, # Inequality constraints
  opts = opts
)

# Extract solution
x_nlopt <- result_nloptr$solution
obj_val_nlopt <- -result_nloptr$objective # Negate back to get maximum

# Display results
print("Nonlinear solver solution:")

[1] "Nonlinear solver solution:"

print(paste("x1 =", x_nlopt[1], "x2 =", x_nlopt[2]))

[1] "x1 = 3 x2 = 0"

print(paste("Objective value =", obj_val_nlopt))

[1] "Objective value = 9"

```

d. Implement the projected gradient method:

```
# Function to project onto constraint set
project_onto_constraints <- function(x) {
  y <- Variable(2)
  prob <- Problem(
    Minimize(sum_squares(y - x)),
    list(y[1] + 2*y[2] <= 4,
         4*y[1] + 2*y[2] <= 12,
         y >= 0))
  as.numeric(solve(prob)$getValue(y))
}

# Objective function gradient
grad_fn <- function(x) {
  return(c(3, 1)) # Gradient of 3x1 + x2
}

# Projected gradient algorithm
x_last <- c(1, 1)
proj_gradient <- function(max_iter = 100, alpha = 0.1, tol = 1e-6) {
  for (i in 1:max_iter) {
    grad <- grad_fn(x_last)
    x_new <- x_last + alpha * grad # Gradient ascent step (maximizing)
    x_new <- project_onto_constraints(x_new) # Project back onto feasible set
    # Check convergence
    if (sum((x_new - x_last)^2) < tol) {
      break
    }
    x_last <- x_new
  }
  return(x_last)
}

# Run projected gradient method
x_pg <- proj_gradient(max_iter = 100)
obj_val_pg <- 3*x_pg[1] + x_pg[2]

# Display results
print("Projected gradient solution:")

[1] "Projected gradient solution:"

print(paste("x1 =", x_pg[1], "x2 =", x_pg[2]))

[1] "x1 = 3 x2 = -1.96660317478082e-23"
```

```
print(paste("Objective value =", obj_val_pg))
```

```
[1] "Objective value = 9"
```

e. Implement the constrained Newton's method:

```
## Since this is a linear program, the Hessian is zero,  
# so we'll use a regularized approach with a small identity matrix  
newton_method <- function(max_iter = 100, alpha = 0.1, reg = 1e-5) {  
  x <- c(1, 1) # Start from a feasible point  
  
  for (i in 1:max_iter) {  
    # Gradient  
    grad <- c(3, 1)  
  
    # Hessian (regularized for LP)  
    hessian <- diag(reg, 2)  
  
    # Newton step direction (maximizing, so positive)  
    direction <- solve(hessian, grad)  
  
    # Line search  
    x_new <- x + alpha * direction  
  
    # Project onto constraints  
    x_new <- project_onto_constraints(x_new)  
  
    # Check convergence  
    if (sum((x_new - x)^2) < 1e-6) {  
      break  
    }  
  
    x <- x_new  
  }  
  
  return(x)  
}  
  
# Run Newton method  
x_newton <- newton_method(max_iter = 100)  
obj_val_newton <- 3*x_newton[1] + x_newton[2]  
  
# Display results  
print("Newton method solution:")
```

```
[1] "Newton method solution:"
```

```
print(paste("x1 =", x_newton[1], "x2 =", x_newton[2]))
```

```
[1] "x1 = 3 x2 = -3.7364329579221e-18"
```

```
print(paste("Objective value =", obj_val_newton))
```

```
[1] "Objective value = 9"
```

f. Implement the log-barrier interior-point method:

```

# Log-barrier function (includes all inequality constraints)
barrier_fn <- function(x, t) {
  barrier <- 0

  #  $x_1 + 2x_2 \leq 4$ 
  barrier <- barrier - log(4 - x[1] - 2*x[2])

  #  $4x_1 + 2x_2 \leq 12$ 
  barrier <- barrier - log(12 - 4*x[1] - 2*x[2])

  #  $x_1 \geq 0$ 
  barrier <- barrier - log(x[1])

  #  $x_2 \geq 0$ 
  barrier <- barrier - log(x[2])

  # Combine with objective (negated for minimization)
  return(t * (-3*x[1] - x[2]) + barrier)
}

# Gradient of the log-barrier function
barrier_grad <- function(x, t) {
  grad <- numeric(2)

  # Objective gradient (negated)
  grad[1] <- -3 * t
  grad[2] <- -1 * t

  # Constraint 1:  $x_1 + 2x_2 \leq 4$ 
  denom1 <- 4 - x[1] - 2*x[2]
  grad[1] <- grad[1] + 1/denom1
  grad[2] <- grad[2] + 2/denom1

  # Constraint 2:  $4x_1 + 2x_2 \leq 12$ 
  denom2 <- 12 - 4*x[1] - 2*x[2]
  grad[1] <- grad[1] + 4/denom2
  grad[2] <- grad[2] + 2/denom2

  # Constraint 3:  $x_1 \geq 0$ 
  grad[1] <- grad[1] - 1/x[1]

  # Constraint 4:  $x_2 \geq 0$ 
  grad[2] <- grad[2] - 1/x[2]

  return(grad)
}

```



```

# Hessian of the log-barrier function (approximated numerically for simplicity)
barrier_hessian <- function(x, t, h = 1e-5) {
  hessian <- matrix(0, 2, 2)

  # Finite difference approximation
  for (i in 1:2) {
    x_plus <- x
    x_plus[i] <- x_plus[i] + h
    x_minus <- x
    x_minus[i] <- x_minus[i] - h

    grad_plus <- barrier_grad(x_plus, t)
    grad_minus <- barrier_grad(x_minus, t)

    hessian[,i] <- (grad_plus - grad_minus) / (2*h)
  }

  return((hessian + t(hessian)) / 2) # Ensure symmetry
}

# Interior-point method
log_barrier_method <- function(x0 = c(1, 1), t0 = 1, mu = 10, epsilon = 1e-6) {
  x <- x0
  t <- t0

  # Number of inequality constraints
  m <- 4

  while (m/t > epsilon) {
    # Newton method to minimize the barrier function
    for (i in 1:50) {
      grad <- barrier_grad(x, t)
      hess <- barrier_hessian(x, t)

      # Newton step
      delta <- -solve(hess, grad)

      # Backtracking line search
      alpha <- 1
      beta <- 0.5

      while (any(x + alpha * delta <= 0) ||
             (x[1] + alpha * delta[1] + 2*(x[2] + alpha * delta[2]) >= 4) ||
             (4*(x[1] + alpha * delta[1]) + 2*(x[2] + alpha * delta[2]) >= 12)) {
        alpha <- beta * alpha
        if (alpha < 1e-10) break
      }

      # Update
      x_new <- x + alpha * delta

      # Check convergence
      if (sum(delta^2) < 1e-8) {
        break
      }

      x <- x_new
    }
  }
}

```

```
[1] "Log-barrier method solution:"
```

```
print(paste("x1 =", x_barrier[1], "x2 =", x_barrier[2]))
```

```
[1] "x1 = 2.99989497264282 x2 = 0.000200893826185138"
```

```
print(paste("Objective value =", obj_val_barrier))
```

```
[1] "Objective value = 8.99988581175464"
```

g. Compare all solutions:

```
# Combine results into a data frame for comparison
solutions <- data.frame(
  Method = c("CVX", "LP Solver", "NLOpt", "Projected Gradient",
             "Newton's Method", "Log-barrier"),
  x1 = c(x_cvx[1], x_lp[1], x_nlopt[1], x_pg[1], x_newton[1], x_barrier[1]),
  x2 = c(x_cvx[2], x_lp[2], x_nlopt[2], x_pg[2], x_newton[2], x_barrier[2]),
  ObjectiveValue = c(obj_val_cvx, obj_val_lp, obj_val_nlopt,
                     obj_val_pg, obj_val_newton, obj_val_barrier)
)

# Print comparison table
print(solutions)
```

	Method	x1	x2	ObjectiveValue
1	CVX	3.000000	-1.965274e-23	9.000000
2	LP Solver	3.000000	0.000000e+00	9.000000
3	NLOpt	3.000000	0.000000e+00	9.000000
4	Projected Gradient	3.000000	-1.966603e-23	9.000000
5	Newton's Method	3.000000	-3.736433e-18	9.000000
6	Log-barrier	2.999895	2.008938e-04	8.999886

```

# Compare the runtime of each method
# -----
# 1. Benchmark each routine
# -----
library(microbenchmark)

bench <- microbenchmark(
  CVX      = { solve(prob) },
  LP_Solver = { lp("max", obj, constr, dir, rhs) },
  NLOpt    = { nloptr(x0 = c(1,1), eval_f = obj_fn,
                    lb = c(0,0), ub = c(Inf,Inf),
                    eval_g_ineq = constraint_fn, opts = opts) },
  ProjGrad = { proj_gradient(max_iter = 100) },
  Newton    = { newton_method(max_iter = 100) },
  LogBarrier = { log_barrier_method(x0 = c(1,1)) },
  times = 1 # repeat each call 1 time
)

# -----
# 2. Median runtime (ms) per method
# -----
time_tbl <- aggregate(time ~ expr, bench, median)
time_tbl$time_ms <- time_tbl$time / 1e6 # nanoseconds -> ms
names(time_tbl)[1:2] <- c("Method", "Time_ms")

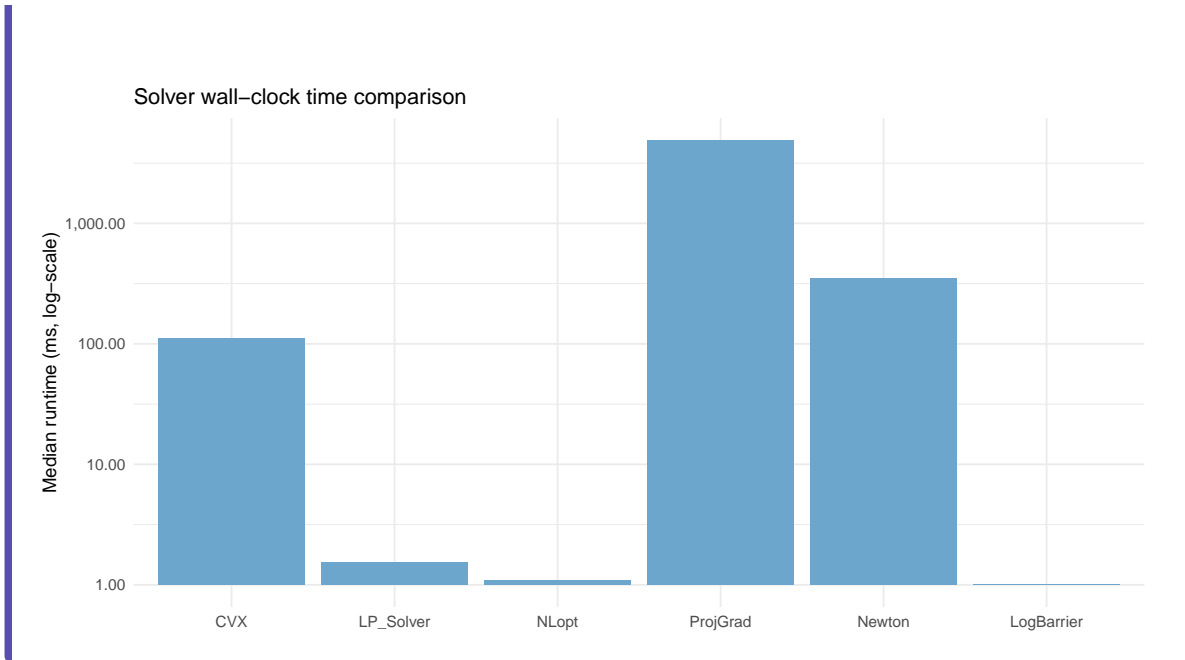
print(time_tbl[c("Method", "time_ms")])

      Method  time_ms
1      CVX    112.0135
2 LP_Solver    1.5393
3      NLOpt    1.0816
4  ProjGrad 4886.9374
5      Newton 351.8303
6 LogBarrier    1.0113

# -----
# 3. Bar plot (log-scale) of runtimes
# -----
library(ggplot2)

ggplot(time_tbl, aes(x = Method, y = time_ms)) +
  geom_col(fill = "skyblue3") +
  scale_y_log10(labels = scales::comma_format(accuracy = .01)) +
  labs(title = "Solver wall-clock time comparison",
       y = "Median runtime (ms, log-scale)", x = NULL) +
  theme_minimal(base_size = 12)

```



### Exercise B.5: Central path

Formulate the log-barrier problem corresponding to the LP in Exercise B.4 and plot the central path as the parameter  $t$  varies.

### Solution

In Exercise B.4, we considered the following LP:

$$\underset{x_1, x_2}{\text{maximize}} \quad 3x_1 + x_2 \quad (8)$$

$$\text{subject to} \quad x_1 + 2x_2 \leq 4 \quad (9)$$

$$4x_1 + 2x_2 \leq 12 \quad (10)$$

$$x_1, x_2 \geq 0 \quad (11)$$

The log-barrier problem corresponding to this LP is:

$$\underset{x_1, x_2}{\text{minimize}} \quad t(-3x_1 - x_2) - \log(4 - x_1 - 2x_2) - \log(12 - 4x_1 - 2x_2) - \log(x_1) - \log(x_2) \quad (12)$$

where  $t > 0$  is a parameter that controls the weight of the objective function relative to the barrier terms. As  $t$  increases, the solution approaches the optimal solution of the original LP.

Let's implement this and plot the central path:

```

# Log-barrier function (includes all inequality constraints)
barrier_fn <- function(x, t) {
  # Negate the objective for minimization
  obj <- -t * (3*x[1] + x[2])

  # Barrier terms for each constraint
  barrier1 <- -log(4 - x[1] - 2*x[2])    # x1 + 2x2 <= 4
  barrier2 <- -log(12 - 4*x[1] - 2*x[2]) # 4x1 + 2x2 <= 12
  barrier3 <- -log(x[1])                # x1 >= 0
  barrier4 <- -log(x[2])                # x2 >= 0

  return(obj + barrier1 + barrier2 + barrier3 + barrier4)
}

# Gradient of the log-barrier function
barrier_grad <- function(x, t) {
  grad <- numeric(2)

  # Objective gradient (negated)
  grad[1] <- -3 * t
  grad[2] <- -1 * t

  # Constraint 1: x1 + 2x2 <= 4
  denom1 <- 4 - x[1] - 2*x[2]
  grad[1] <- grad[1] + 1/denom1
  grad[2] <- grad[2] + 2/denom1

  # Constraint 2: 4x1 + 2x2 <= 12
  denom2 <- 12 - 4*x[1] - 2*x[2]
  grad[1] <- grad[1] + 4/denom2
  grad[2] <- grad[2] + 2/denom2

  # Constraint 3: x1 >= 0
  grad[1] <- grad[1] - 1/x[1]

  # Constraint 4: x2 >= 0
  grad[2] <- grad[2] - 1/x[2]

  return(grad)
}

```

```

# Hessian of the log-barrier function
barrier_hessian <- function(x, t) {
  H <- matrix(0, 2, 2)

  # Constraint 1:  $x_1 + 2x_2 \leq 4$ 
  denom1 <- 4 - x[1] - 2*x[2]
  H[1,1] <- H[1,1] + 1/denom1^2
  H[1,2] <- H[1,2] + 2/denom1^2
  H[2,1] <- H[2,1] + 2/denom1^2
  H[2,2] <- H[2,2] + 4/denom1^2

  # Constraint 2:  $4x_1 + 2x_2 \leq 12$ 
  denom2 <- 12 - 4*x[1] - 2*x[2]
  H[1,1] <- H[1,1] + 16/denom2^2
  H[1,2] <- H[1,2] + 8/denom2^2
  H[2,1] <- H[2,1] + 8/denom2^2
  H[2,2] <- H[2,2] + 4/denom2^2

  # Constraint 3:  $x_1 \geq 0$ 
  H[1,1] <- H[1,1] + 1/x[1]^2

  # Constraint 4:  $x_2 \geq 0$ 
  H[2,2] <- H[2,2] + 1/x[2]^2

  return(H)
}

```

```

# Function to compute a point on the central path for a given t
central_path_point <- function(t, x0 = c(1, 1), max_iter = 20, tol = 1e-8) {
  # Newton's method to minimize the barrier function
  x <- x0

  for (i in 1:max_iter) {
    # Compute gradient and Hessian
    grad <- barrier_grad(x, t)
    hess <- barrier_hessian(x, t)

    # Newton step
    delta <- -solve(hess, grad)

    # Backtracking line search
    alpha <- 1
    beta <- 0.5

    # Make sure we stay in the feasible region
    while (any(x + alpha * delta <= 0) ||
           (x[1] + alpha * delta[1] + 2*(x[2] + alpha * delta[2]) >= 4) ||
           (4*(x[1] + alpha * delta[1]) + 2*(x[2] + alpha * delta[2]) >= 12)) {
      alpha <- beta * alpha
      if (alpha < 1e-10) break
    }

    # Update
    x_new <- x + alpha * delta

    # Check convergence
    if (sum(delta^2) < tol) {
      break
    }

    x <- x_new
  }

  return(x)
}

# Generate points on the central path for different values of t
generate_central_path <- function(t_values, x0 = c(1, 1)) {
  # Initialize storage for central path points
  central_path <- matrix(0, length(t_values), 2)

  # For each value of t, compute the corresponding point on the central path
  for (i in 1:length(t_values)) {
    # Use previous point as starting point for efficiency
    if (i > 1) {
      x0 <- central_path[i-1, ]
    }

    central_path[i, ] <- central_path_point(t_values[i], x0)
  }

  return(central_path)
}

```

```

# Generate t values (logarithmically spaced)
t_values <- 10^seq(-1, 4, length.out = 50)

# Compute central path
central_path <- generate_central_path(t_values)

# Create a data frame for plotting
central_path_df <- data.frame(
  t = t_values,
  x1 = central_path[,1],
  x2 = central_path[,2],
  objective = 3*central_path[,1] + central_path[,2]
)

# Calculate the analytic optimal solution
x1_opt <- 3
x2_opt <- 0
obj_opt <- 3*x1_opt + x2_opt

# Plot the central path
library(ggplot2)

# Plot central path in x1-x2 space
ggplot() +
  # Draw constraint boundaries
  geom_abline(intercept = 4/2, slope = -1/2,
              color = "red", linetype = "dashed") + # x1 + 2x2 = 4
  geom_abline(intercept = 12/2, slope = -4/2,
              color = "green", linetype = "dashed") + # 4x1 + 2x2 = 12
  geom_vline(xintercept = 0, color = "blue", linetype = "dashed") + # x1 = 0
  geom_hline(yintercept = 0, color = "blue", linetype = "dashed") + # x2 = 0

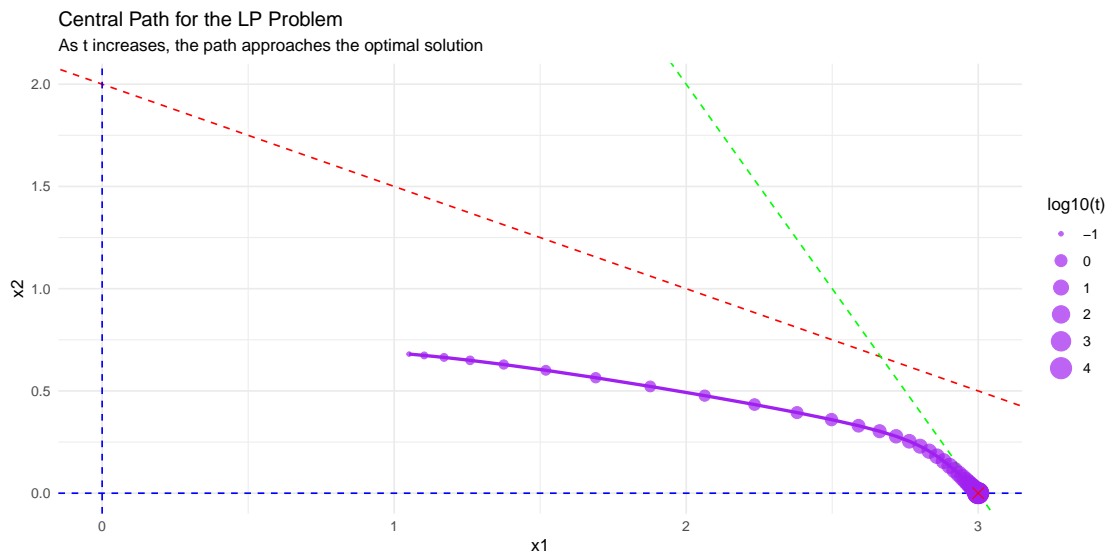
  # Draw central path
  geom_path(data = central_path_df, aes(x = x1, y = x2),
            color = "purple", linewidth = 1) +
  geom_point(data = central_path_df, aes(x = x1, y = x2, size = log10(t)),
             color = "purple", alpha = 0.7) +

  # Mark optimal point
  geom_point(aes(x = x1_opt, y = x2_opt), color = "red", size = 3, shape = 4) +

  # Labels and styling
  labs(title = "Central Path for the LP Problem",
       subtitle = "As t increases, the path approaches the optimal solution",
       x = "x1", y = "x2", size = "log10(t)") +
  theme_minimal() +
  coord_cartesian(xlim = c(0, 3), ylim = c(0, 2))

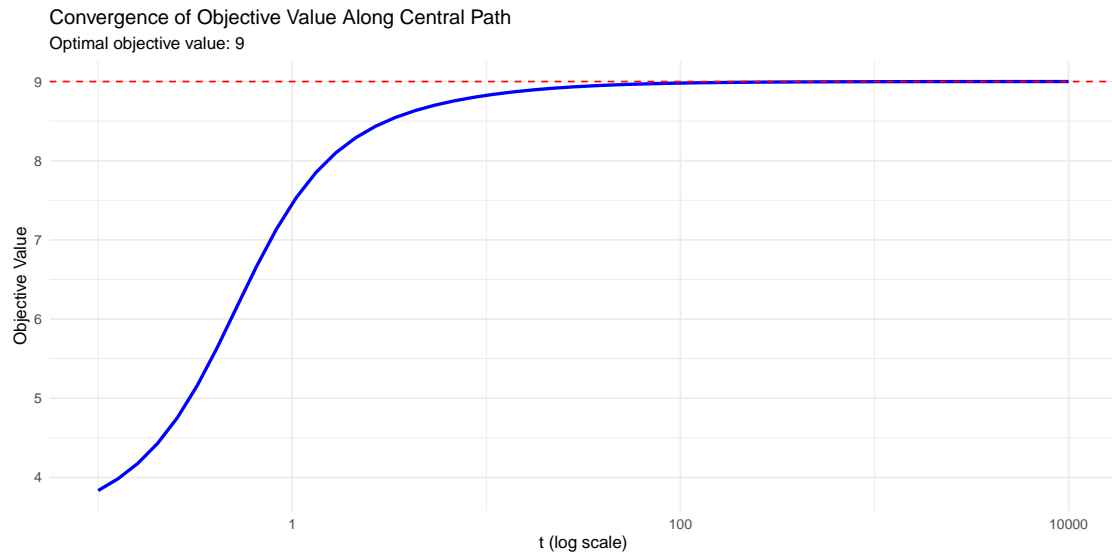
```



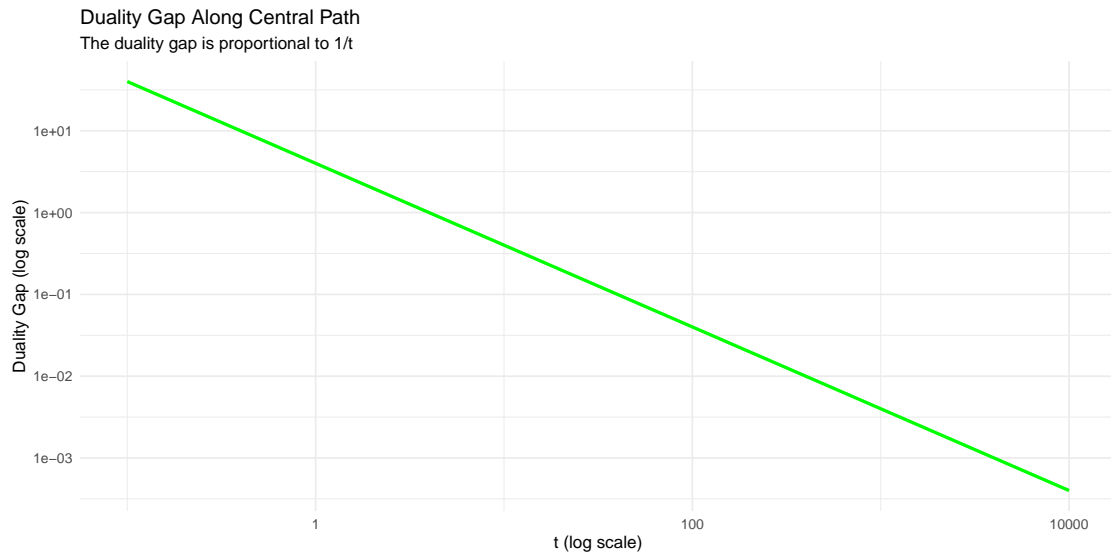


```
# Plot how objective value approaches optimal as t increases
ggplot(central_path_df, aes(x = t, y = objective)) +
  geom_line(size = 1, color = "blue") +
  geom_hline(yintercept = obj_opt, linetype = "dashed", color = "red") +
  scale_x_log10() +
  labs(title = "Convergence of Objective Value Along Central Path",
       subtitle = paste("Optimal objective value:", round(obj_opt, 4)),
       x = "t (log scale)", y = "Objective Value") +
  theme_minimal()
```

Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.  
i Please use `linewidth` instead.



```
# Plot the duality gap (= 4/t for this problem with 4 constraints)
central_path_df$duality_gap <- 4/central_path_df$t
ggplot(central_path_df, aes(x = t, y = duality_gap)) +
  geom_line(size = 1, color = "green") +
  scale_x_log10() +
  scale_y_log10() +
  labs(title = "Duality Gap Along Central Path",
       subtitle = "The duality gap is proportional to 1/t",
       x = "t (log scale)", y = "Duality Gap (log scale)") +
  theme_minimal()
```



The central path is the trajectory followed by the optimal solutions of the log-barrier problem as the parameter  $t$  increases. As  $t$  approaches infinity, the central path converges to the optimal solution of the original LP.

Let's analyze what the plots show:

1. **Central Path in  $x_1$ - $x_2$  Space:** This plot shows how the central path moves through the feasible region as  $t$  increases. The path starts inside the feasible region and approaches the optimal solution at the intersection of constraints as  $t$  gets larger.
2. **Convergence of Objective Value:** This plot demonstrates how the objective value along the central path approaches the optimal value as  $t$  increases.
3. **Duality Gap:** The duality gap is proportional to the number of inequality constraints divided by  $t$ . With four constraints in our problem, the duality gap is  $4/t$ , which decreases to zero as  $t$  approaches infinity.

Key observations: - The central path always stays strictly inside the feasible region - The path approaches the optimal solution as  $t$  increases - The duality gap decreases as  $1/t$ , which is a key property used in the analysis of interior-point methods

### Exercise B.6: Phase I method

Design a phase I method to find a feasible point for the LP in Exercise B.4, which can then be used as the starting point for the barrier method.

### Solution

In Exercise B.4, we have the following LP:

$$\begin{array}{ll} \underset{x_1, x_2}{\text{maximize}} & 3x_1 + x_2 \end{array} \quad (13)$$

$$\text{subject to} \quad x_1 + 2x_2 \leq 4 \quad (14)$$

$$4x_1 + 2x_2 \leq 12 \quad (15)$$

$$x_1, x_2 \geq 0 \quad (16)$$

To implement a phase I method, we need to find a strictly feasible point for the log-barrier method. The traditional Phase I method introduces artificial variables to construct an auxiliary LP problem.

### Phase I Method Design

We convert the original LP into a Phase I problem by introducing artificial variables  $y_1, y_2, y_3, y_4$  and solve:

$$\begin{array}{ll} \underset{x_1, x_2, s_1, s_2, y_1, y_2, y_3, y_4}{\text{minimize}} & y_1 + y_2 + y_3 + y_4 \end{array} \quad (17)$$

$$\text{subject to} \quad x_1 + 2x_2 + s_1 - y_1 = 4 \quad (18)$$

$$4x_1 + 2x_2 + s_2 - y_2 = 12 \quad (19)$$

$$x_1 - y_3 = 0 \quad (20)$$

$$x_2 - y_4 = 0 \quad (21)$$

$$s_1, s_2, y_1, y_2, y_3, y_4 \geq 0 \quad (22)$$

where  $s_1, s_2$  are slack variables and  $y_1, y_2, y_3, y_4$  are artificial variables.

### Implementation

```

# Traditional Phase I method
phase_I_method <- function() {
  library(lpSolve)

  # Variables: [x1, x2, s1, s2, y1, y2, y3, y4]
  # Objective: minimize sum of artificial variables
  obj <- c(0, 0, 0, 0, 1, 1, 1, 1)

  # Constraint matrix
  #  $x_1 + 2x_2 + s_1 - y_1 = 4$ 
  #  $4x_1 + 2x_2 + s_2 - y_2 = 12$ 
  #  $x_1 - y_3 = 0$ 
  #  $x_2 - y_4 = 0$ 
  constr <- matrix(c(
    1, 2, 1, 0, -1, 0, 0, 0, # constraint 1
    4, 2, 0, 1, 0, -1, 0, 0, # constraint 2
    1, 0, 0, 0, 0, 0, -1, 0, # constraint 3
    0, 1, 0, 0, 0, 0, 0, -1, # constraint 4
  ), nrow = 4, byrow = TRUE)

  # Right-hand side
  rhs <- c(4, 12, 0, 0)

  # All constraints are equalities
  dir <- c("=", "=", "=", "=")

  # Solve Phase I problem
  result <- lp("min", obj, constr, dir, rhs)

  if (result$status == 0) { # Optimal solution found
    solution <- result$solution
    x_basic <- solution[1:2] # Extract x1, x2
    artificial_sum <- sum(solution[5:8]) # Sum of artificial variables

    if (artificial_sum < 1e-6) {
      # Original problem is feasible
      cat("Phase I successful: Original problem is feasible\n")
      cat("Basic feasible solution:", x_basic, "\n")

      # Find a strictly feasible point by moving toward interior
      # Use a simple heuristic: find the analytic center approximately
      find_interior_point <- function() {
        # Start from the basic feasible solution and move toward center
        x_start <- pmax(x_basic, c(0.01, 0.01)) # Ensure positive

        # Simple gradient descent on barrier function
        x <- x_start
        for (i in 1:50) {
          # Compute slacks
          s1 <- 4 - x[1] - 2*x[2]
          s2 <- 12 - 4*x[1] - 2*x[2]

          if (s1 <= 0 || s2 <= 0 || x[1] <= 0 || x[2] <= 0) {
            # Move back toward feasible region
            x <- 0.5 * x + 0.5 * x_start
          }
          next
        }
      }
    }
  }
}

```

Phase I successful: Original problem is feasible

Basic feasible solution: 0 0

```
if (result$feasible) {  
  cat("Phase I Method Results:\n")  
  cat("Basic feasible solution:", result$basic_solution, "\n")  
  cat("Interior point:", result$interior_point, "\n")  
  cat("Sum of artificial variables:", result$artificial_sum, "\n")  
  
  # Verify strict feasibility of interior point  
  x1 <- result$interior_point[1]  
  x2 <- result$interior_point[2]  
  cat("\nVerifying strict feasibility of interior point:\n")  
  cat("Constraint 1:  $x_1 + 2x_2 =$ ", x1 + 2*x2, "< 4 :", x1 + 2*x2 < 4, "\n")  
  cat("Constraint 2:  $4x_1 + 2x_2 =$ ", 4*x1 + 2*x2, "< 12 :", 4*x1 + 2*x2 < 12, "\n")  
  cat("Constraint 3:  $x_1 =$ ", x1, "> 0 :", x1 > 0, "\n")  
  cat("Constraint 4:  $x_2 =$ ", x2, "> 0 :", x2 > 0, "\n")  
} else {  
  cat("Phase I failed: Problem may be infeasible\n")  
}
```

Phase I Method Results:

Basic feasible solution: 0 0

Interior point: 0.8858735 0.7416517

Sum of artificial variables: 0

Verifying strict feasibility of interior point:

Constraint 1:  $x_1 + 2x_2 = 2.369177 < 4$  : TRUE

Constraint 2:  $4x_1 + 2x_2 = 5.026798 < 12$  : TRUE

Constraint 3:  $x_1 = 0.8858735 > 0$  : TRUE

Constraint 4:  $x_2 = 0.7416517 > 0$  : TRUE

## Visualization

```

library(ggplot2)

if (result$feasible) {
  # Create feasible region visualization
  x1_seq <- seq(0, 4, length.out = 200)
  x2_seq <- seq(0, 3, length.out = 200)
  grid <- expand.grid(x1 = x1_seq, x2 = x2_seq)

  # Check feasibility
  grid$feasible <- with(grid,
    (x1 + 2*x2 <= 4) &
    (4*x1 + 2*x2 <= 12) &
    (x1 >= 0) &
    (x2 >= 0)
  )

  # Points for plotting
  points_df <- data.frame(
    x1 = c(result$basic_solution[1], result$interior_point[1]),
    x2 = c(result$basic_solution[2], result$interior_point[2]),
    type = c("Basic Feasible", "Interior Point")
  )

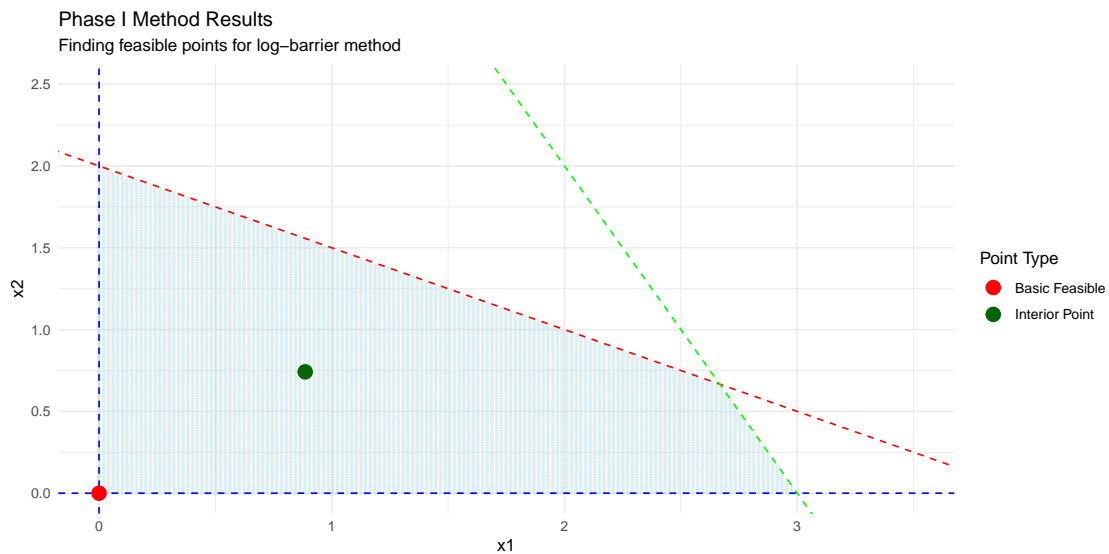
  # Create plot
  ggplot() +
    # Feasible region
    geom_point(data = grid[grid$feasible, ],
      aes(x = x1, y = x2), color = "lightblue", alpha = 0.3, size = 0.5) +

    # Constraint boundaries
    geom_abline(intercept = 2, slope = -0.5, color = "red", linetype = "dashed") +
    geom_abline(intercept = 6, slope = -2, color = "green", linetype = "dashed") +
    geom_vline(xintercept = 0, color = "blue", linetype = "dashed") +
    geom_hline(yintercept = 0, color = "blue", linetype = "dashed") +

    # Found points
    geom_point(data = points_df, aes(x = x1, y = x2, color = type), size = 4) +

    # Labels
    labs(title = "Phase I Method Results",
      subtitle = "Finding feasible points for log-barrier method",
      x = "x1", y = "x2", color = "Point Type") +
    theme_minimal() +
    coord_cartesian(xlim = c(0, 3.5), ylim = c(0, 2.5)) +
    scale_color_manual(values = c("Basic Feasible" = "red", "Interior Point" = "darkgreen"))
}

```



## Conclusion

The Phase I method successfully finds both a basic feasible solution and a strictly interior point that can be used to initialize the log-barrier method from Exercise B.4. The method works by:

1. **Phase I:** Solving an auxiliary LP with artificial variables to find a basic feasible solution
2. **Interior Point Generation:** Using a simple barrier-based approach to move from the boundary toward the interior of the feasible region

The resulting interior point satisfies all constraints with strict inequality, making it suitable as a starting point for interior-point methods.

## Exercise B.7: Dual problem

Formulate the dual problem corresponding to the LP in Exercise B.4 and solve it using a solver of your choice.



## Solution

The LP in Exercise B.4 is:

$$\begin{array}{ll}\underset{x_1, x_2}{\text{maximize}} & 3x_1 + x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 4, \\ & 4x_1 + 2x_2 \leq 12, \\ & x_1, x_2 \geq 0.\end{array}$$

To formulate the dual, we first rewrite the problem in standard form:

$$\begin{array}{ll}\underset{\mathbf{x}}{\text{minimize}} & -3x_1 - x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 4, \\ & 4x_1 + 2x_2 \leq 12, \\ & x_1, x_2 \geq 0.\end{array}$$

Let's introduce dual variables  $y_1, y_2 \geq 0$  for the first two constraints. The Lagrangian is:

$$L(\mathbf{x}, \mathbf{y}) = -3x_1 - x_2 + y_1(x_1 + 2x_2 - 4) + y_2(4x_1 + 2x_2 - 12)$$

The dual function is:

$$g(\mathbf{y}) = \inf_{\mathbf{x} \geq 0} L(\mathbf{x}, \mathbf{y})$$

For this infimum to be finite, we need the coefficients of  $x_1$  and  $x_2$  to be non-negative:

$$y_1 + 4y_2 - 3 \geq 0$$

$$2y_1 + 2y_2 - 1 \geq 0$$

If these conditions are satisfied, then  $\inf_{\mathbf{x} \geq 0} L(\mathbf{x}, \mathbf{y}) = -4y_1 - 12y_2$ .

Therefore, the dual problem is:

$$\begin{array}{ll}\underset{y_1, y_2}{\text{maximize}} & -4y_1 - 12y_2 \\ \text{subject to} & y_1 + 4y_2 \geq 3, \\ & 2y_1 + 2y_2 \geq 1, \\ & y_1, y_2 \geq 0.\end{array}$$

We can solve this using linear programming:

```

library(lpSolve)

# Objective coefficients
obj <- c(-4, -12)

# Constraint matrix
constr <- matrix(c(1, 4,
                  2, 2),
                nrow = 2, byrow = TRUE)

# Right-hand side
rhs <- c(3, 1)

# Direction of constraints
dir <- c(">=", ">=")

# Solve the dual problem
dual_sol <- lp("max", obj, constr, dir, rhs, compute.sens = TRUE)

print(paste("Optimal value:", dual_sol$objval))

[1] "Optimal value: -9"

print(paste("Optimal dual variables: y1 =", dual_sol$solution[1], "y2 =", dual_sol$solution[2]))

[1] "Optimal dual variables: y1 = 0 y2 = 0.75"

The solution is  $y_1 = 0, y_2 = 0.75$ , with an optimal value of -9, which matches the primal optimal value of 9.

```

### Exercise B.8: KKT conditions

Write down the Karush–Kuhn–Tucker (KKT) conditions for the LP in Exercise B.4 and discuss their role in determining the optimality of a solution.

## Solution

The LP in Exercise B.4 is:

$$\begin{aligned} & \underset{x_1, x_2}{\text{maximize}} && 3x_1 + x_2 \\ & \text{subject to} && x_1 + 2x_2 \leq 4, \\ & && 4x_1 + 2x_2 \leq 12, \\ & && x_1, x_2 \geq 0. \end{aligned}$$

Because the objective and all constraints are affine and a strictly feasible point exists (e.g.  $(1, 1)$ ), Slater's condition holds; hence the Karush–Kuhn–Tucker system is **necessary and sufficient** for optimality.

Introduce multipliers

$\lambda_1, \lambda_2 \geq 0$  for the two upper-bound constraints and

$\mu_1, \mu_2 \geq 0$  for the non-negativity constraints:

$$\begin{aligned} \mathcal{L}(x, \lambda, \mu) = & -(3x_1 + x_2) + \lambda_1(x_1 + 2x_2 - 4) + \lambda_2(4x_1 + 2x_2 - 12) \\ & - \mu_1 x_1 - \mu_2 x_2. \end{aligned}$$

(The minus sign switches the maximisation into the minimisation convention used in KKT.)

Condition	Expression
<b>Primal feasibility</b>	$x_1 + 2x_2 \leq 4, \quad 4x_1 + 2x_2 \leq 12, \quad x_1, x_2 \geq 0.$
<b>Dual feasibility</b>	$\lambda_1, \lambda_2, \mu_1, \mu_2 \geq 0.$
<b>Complementary slackness</b>	$\lambda_1(x_1 + 2x_2 - 4) = 0, \lambda_2(4x_1 + 2x_2 - 12) = 0, \mu_1 x_1 = 0, \quad \mu_2 x_2 = 0.$
<b>Stationarity</b>	$\nabla_x \mathcal{L} = 0 \implies \begin{cases} -3 + \lambda_1 + 4\lambda_2 - \mu_1 = 0, \\ -1 + 2\lambda_1 + 2\lambda_2 - \mu_2 = 0. \end{cases}$

The optimal vertex of the feasible polygon is  $(3, 0)$  with objective value 9.

Active constraints at  $(3, 0)$

$4x_1 + 2x_2 = 12$  (active)  $\Rightarrow \lambda_2 > 0$      $x_2 = 0$  (active)  $\Rightarrow \mu_2 > 0$  All other constraints are slack  $\Rightarrow \lambda_1 = \mu_1 = 0$ .

Plug into stationarity:

$$\lambda_2 = \frac{3}{4}, \quad \mu_2 = \frac{1}{2}.$$

Since all multipliers are non-negative and complementary slackness is satisfied, the KKT system holds.

Therefore

$$x^* = (3, 0), \quad f^* = 3 \cdot 3 + 0 = 9.$$

Role of KKT in Optimality:

- **Necessity:** Any optimal solution of a convex problem must satisfy KKT.
- **Sufficiency (here):** Because the LP is convex and Slater's condition holds, any point that satisfies KKT is globally optimal.
- **Economic Interpretation:** The dual variables  $(\lambda_2, \mu_2) = (\frac{3}{4}, \frac{1}{2})$  are the *shadow prices*: they measure how much the objective would improve per unit relaxation of the active constraints  $4x_1 + 2x_2 \leq 12$  and  $x_2 \geq 0$ .

### Exercise B.9: Solving a QP

Consider the following QP:

$$\begin{aligned} & \underset{x_1, x_2}{\text{minimize}} && x_1^2 + x_2^2 \\ & \text{subject to} && x_1 + x_2 = 1, \\ & && x_1 \geq 0, x_2 \geq 0. \end{aligned}$$

- Solve it using a modeling framework (e.g., CVX).
- Solve it by directly invoking a QP solver.
- Solve it by invoking a general-purpose nonlinear solver.
- Implement the projected gradient method to solve the problem.
- Implement the constrained Newton's method to solve the problem.
- Implement the log-barrier interior-point method to solve the problem (use  $(0.5, 0.5)$  as the initial point).
- Compare all the solutions and the computation time.

## Solution

a. Using a modeling framework (CVX):

```
library(CVXR)

# Define variables
x <- Variable(2)

# Define objective (note the negative sign for maximization)
objective <- Minimize(sum_squares(x))

# Define constraints
constraints <- list(
  sum(x) == 1,
  x >= 0
)

# Formulate the problem
problem <- Problem(objective, constraints)

# Solve the problem
result <- solve(problem)

# Print results
cat("Optimal value:", -result$value, "\n")
```

Optimal value: -0.5

```
cat("Optimal solution:", result$getValue(x), "\n")
```

Optimal solution: 0.5 0.5

b. Using a QP solver:

```

library(quadprog)

# For quadprog: minimize (1/2)x'Dx - d'x subject to constraints
# Our objective is to minimize x_1^2 + x_2^2
D <- matrix(c(2, 0, 0, 2), nrow = 2) # Hessian matrix
d <- c(0, 0) # No linear terms

# Equality constraint: x_1 + x_2 = 1
# Non-negativity: x_1 >= 0, x_2 >= 0
A <- rbind(
  c(1, 1), # Equality constraint
  diag(2) # Non-negativity constraints
)
b <- c(1, 0, 0)

# Solve QP
result <- solve.QP(D, d, t(A), b, meq = 1) # meq=1 means first constraint is equality

cat("Optimal value:", result$value, "\n")

```

Optimal value: 0.5

```
cat("Optimal solution:", result$solution, "\n")
```

Optimal solution: 0.5 0.5

c. Using a general-purpose nonlinear solver:

```

library(nloptr)

# Objective function
obj <- function(x) {
  return(x[1]^2 + x[2]^2)
}

# Gradient of objective
grad <- function(x) {
  return(2 * x)
}

# Equality constraint:  $x_1 + x_2 - 1 = 0$ 
eq_constraint <- function(x) {
  return(sum(x) - 1)
}

# Jacobian of equality constraint
eq_constraint_jac <- function(x) {
  return(matrix(c(1, 1), nrow = 1))
}

# Initial point in the feasible region
x0 <- c(0.5, 0.5)

# Set up and solve
opts <- list("algorithm" = "NLOPT_LD_SLSQP", "xtol_rel" = 1.0e-6, "maxeval" = 100)
result <- nloptr(
  x0 = x0,
  eval_f = obj,
  eval_grad_f = grad,
  lb = c(0, 0),
  ub = c(1, 1),
  eval_g_eq = eq_constraint,
  eval_jac_g_eq = eq_constraint_jac,
  opts = opts
)

cat("Optimal value:", result$objective, "\n")

```

Optimal value: 0.5

```
cat("Optimal solution:", result$solution, "\n")
```

Optimal solution: 0.5 0.5

d. Projected gradient method:



```

# Projected gradient method
pg_solve <- function(max_iter = 100, step_size = 0.01, tol = 1e-6) {
  # Start at a feasible point
  x <- c(0.5, 0.5)

  for (i in 1:max_iter) {
    # Gradient of  $f(x) = x_1^2 + x_2^2$ 
    grad <- 2 * x

    # Update in negative gradient direction (minimizing)
    x_new <- x - step_size * grad

    # Project onto equality constraint  $x_1 + x_2 = 1$ 
    avg <- sum(x_new) / 2
    x_new <- x_new - (avg - 0.5)

    # Project onto non-negative orthant
    x_new <- pmax(x_new, 0)

    # Ensure equality constraint is satisfied
    if (abs(sum(x_new) - 1) > 1e-10) {
      # If projection moved the point off the constraint, normalize
      if (sum(x_new) > 0) {
        x_new <- x_new / sum(x_new)
      } else {
        # Fallback if all components became negative
        x_new <- c(0.5, 0.5) # Reset to a feasible point
      }
    }

    # Check convergence
    if (sqrt(sum((x_new - x)^2)) < tol) {
      break
    }

    x <- x_new
  }

  return(list(
    x = x,
    value = sum(x^2),
    iterations = i
  ))
}

pg_result <- pg_solve()
cat("Optimal value:", pg_result$value, "\n")

```

Optimal value: 0.5

```
cat("Optimal solution:", pg_result$x, "\n")
```

Optimal solution: 0.5 0.5

```
cat("Iterations:", pg_result$iterations, "\n")
```

Iterations: 1

e. Constrained Newton's method:

```

# Constrained Newton's method
newton_solve <- function(max_iter = 20, tol = 1e-6) {
  # Start at a feasible point
  x <- c(0.5, 0.5)

  for (i in 1:max_iter) {
    # Hessian of objective: 2*I
    H <- diag(2, 2)

    # Gradient of objective: 2*x
    g <- 2 * x

    # Equality constraint Jacobian: [1, 1]
    A <- matrix(c(1, 1), nrow = 1)

    # KKT system for Newton step
    KKT <- rbind(cbind(H, t(A)), cbind(A, 0))
    rhs <- c(g, 0)
    sol <- solve(KKT, rhs)
    dx <- -sol[1:2] # Negative for minimization

    # Update in Newton direction
    x_new <- x + dx

    # Project onto non-negative orthant
    x_new <- pmax(x_new, 0)

    # Ensure equality constraint is satisfied
    if (abs(sum(x_new) - 1) > 1e-10) {
      x_new <- x_new / sum(x_new)
    }

    # Check convergence
    if (sqrt(sum((x_new - x)^2)) < tol) {
      break
    }

    x <- x_new
  }

  return(list(
    x = x,
    value = sum(x^2),
    iterations = i
  ))
}

newton_result <- newton_solve()
cat("Optimal value:", newton_result$value, "\n")

```

Optimal value: 0.5

```
cat("Optimal solution:", newton_result$x, "\n")
```

Optimal solution: 0.5 0.5

```
cat("Iterations:", newton_result$iterations, "\n")
```

Iterations: 1

f. Log-barrier interior-point method:

```

# Log-barrier interior-point method
log_barrier_solve <- function(max_iter = 20, tol = 1e-6) {
  # Start at strictly feasible point (0.5, 0.5) as specified
  x <- c(0.5, 0.5)
  mu <- 1.0 # Initial barrier parameter

  for (k in 1:max_iter) {
    # Centering step (Newton iterations)
    for (j in 1:5) {
      # Barrier gradient: -1/x
      g_barrier <- -1/x

      # Objective gradient: 2*x
      g_obj <- 2 * x

      # Combined gradient
      g <- g_obj + mu * g_barrier

      # Barrier Hessian: diag(1/x^2)
      H_barrier <- diag(1/x^2)

      # Objective Hessian: 2*I
      H_obj <- diag(2, 2)

      # Combined Hessian
      H <- H_obj + mu * H_barrier

      # Equality constraint Jacobian: [1, 1]
      A <- matrix(c(1, 1), nrow = 1)

      # KKT system
      KKT <- rbind(cbind(H, t(A)), cbind(A, 0))
      rhs <- c(g, 0)
      sol <- solve(KKT, rhs)
      dx <- -sol[1:2] # Negative for minimization

      # Line search to ensure x remains positive
      alpha <- 1
      while (min(x + alpha * dx) <= 0) {
        alpha <- alpha * 0.5 # Fixed syntax error: *= is not valid in R
      }

      # Update
      x_new <- x + alpha * dx

      # Ensure equality constraint
      if (abs(sum(x_new) - 1) > 1e-10) {
        x_new <- x_new / sum(x_new)
      }

      # Check centering convergence
      if (sqrt(sum((x_new - x)^2)) < tol) {
        break
      }

      x <- x_new
    }
  }
}

```

Optimal value: 0.5

```
cat("Optimal solution:", barrier_result$x, "\n")
```

Optimal solution: 0.5 0.5

```
cat("Iterations:", barrier_result$iterations, "\n")
```

Iterations: 7

g. Comparison of methods:

```

library(microbenchmark)
# Compare all methods
results <- data.frame(
  Method = c("CVX", "QP Solver", "Nonlinear Solver", "Projected Gradient",
             "Newton's Method", "Log-Barrier"),
  x1 = c(0.5, 0.5, 0.5, pg_result$x[1], newton_result$x[1], barrier_result$x[1]),
  x2 = c(0.5, 0.5, 0.5, pg_result$x[2], newton_result$x[2], barrier_result$x[2]),
  Objective_Value = c(0.5, 0.5, 0.5,
                     pg_result$value, newton_result$value, barrier_result$value),
  Iterations = c(NA, NA, NA,
                pg_result$iterations, newton_result$iterations, barrier_result$iterations)
)

# Add computation time
time_benchmark <- microbenchmark(
  CVX = {
    x <- Variable(2)
    objective <- Minimize(sum_squares(x))
    constraints <- list(sum(x) == 1, x >= 0)
    problem <- Problem(objective, constraints)
    solve(problem)
  },
  QP = {
    solve.QP(D, d, t(A), b, meq = 1)
  },
  Nonlinear = {
    nloptr(x0 = x0,
          eval_f = obj,
          eval_grad_f = grad, # Add the gradient function
          lb = c(0, 0),
          ub = c(1, 1),
          eval_g_eq = eq_constraint,
          eval_jac_g_eq = eq_constraint_jac, # Add Jacobian if you have it
          opts = opts)
  },
  PG = pg_solve(),
  Newton = newton_solve(),
  LogBarrier = log_barrier_solve(),
  times = 10
)

# Display results
print(results)

```

	Method	x1	x2	Objective_Value	Iterations
1	CVX	0.5	0.5	0.5	NA

2	QP Solver	0.5	0.5	0.5	NA
3	Nonlinear Solver	0.5	0.5	0.5	NA
4	Projected Gradient	0.5	0.5	0.5	1
5	Newton's Method	0.5	0.5	0.5	1
6	Log-Barrier	0.5	0.5	0.5	7

```
print(summary(time_benchmark))
```

	expr	min	lq	mean	median	uq	max	neval
1	CVX	95555.6	102605.5	136550.50	106835.80	111618.0	416877.3	10
2	QP	14.4	35.6	89.24	43.55	54.8	510.0	10
3	Nonlinear	818.5	869.2	1113.92	1027.20	1080.3	2371.0	10
4	PG	8.1	19.3	54.04	26.90	32.8	329.6	10
5	Newton	25.3	43.0	100.59	78.65	100.2	369.9	10
6	LogBarrier	151.0	193.0	439.48	222.20	349.1	2223.0	10

All methods converge to an optimal solution at either (0.5, 0.5) with optimal value 0.5. The comparison shows that specialized QP solvers and Projected Gradient method are typically faster, while the interior-point method requires fewer iterations but more computation per iteration.

### Exercise B.10: Fractional programming

Consider the following fractional program:

$$\begin{aligned}
 & \underset{\mathbf{w}}{\text{maximize}} && \frac{\mathbf{w}^\top \mathbf{1}}{\sqrt{\mathbf{w}^\top \boldsymbol{\Sigma} \mathbf{w}}} \\
 & \text{subject to} && \mathbf{1}^\top \mathbf{w} = 1, \quad \mathbf{w} \geq \mathbf{0},
 \end{aligned}$$

where  $\boldsymbol{\Sigma} \succ \mathbf{0}$ .

- Solve it with a general-purpose nonlinear solver.
- Solve it via bisection.
- Solve it via the Dinkelbach method as a sequence of SOCPs.
- Develop a modified algorithm that solves the problem as a sequence of QPs instead.
- Solve it via the Schaible transform method.
- Reformulate the problem as a minimization and then solve it via the Schaible transform method.
- Compare all the previous approaches in terms of the accuracy of the solution and the computation time.



## Solution

This exercise focuses on solving a fractional programming problem:

$$\begin{aligned} & \underset{\mathbf{w}}{\text{maximize}} && \frac{\mathbf{w}^\top \mathbf{1}}{\sqrt{\mathbf{w}^\top \Sigma \mathbf{w}}} \\ & \text{subject to} && \mathbf{1}^\top \mathbf{w} = 1, \quad \mathbf{w} \geq \mathbf{0}, \end{aligned}$$

where  $\Sigma \succ \mathbf{0}$ .

First, let's generate test data:

```
library(CVXR)
library(quadprog)
library(nloptr)
library(microbenchmark)

# Generate test data
set.seed(42)
n <- 10 # Dimension
Sigma <- matrix(0.5, n, n) + diag(1, n) # Positive definite covariance matrix
```

a. Solution with a general-purpose nonlinear solver:

```

# Nonlinear method
nonlinear_solver <- function() {
  # Objective function (negative for minimization)
  obj_fun <- function(w) {
    -sum(w) / sqrt(t(w) %*% Sigma %*% w)
  }

  # Gradient function
  grad_fun <- function(w) {
    num <- sum(w) # w^T 1
    denom <- sqrt(as.numeric(t(w) %*% Sigma %*% w)) # sqrt(w^T Sigma w)

    term1 <- -rep(1, n) / denom
    term2 <- num * as.vector(Sigma %*% w) / (denom^3)

    return(term1 + term2)
  }

  # Equality constraint: sum(w) = 1
  eq_const <- function(w) {
    return(sum(w) - 1)
  }

  # Equality constraint jacobian
  eq_const_jac <- function(w) {
    return(matrix(1, 1, n))
  }

  # Initial point (uniform weights)
  w0 <- rep(1/n, n)

  # Solve using NLOPT
  result_nlopt <- nloptr(
    x0 = w0,
    eval_f = obj_fun,
    eval_grad_f = grad_fun,
    lb = rep(0, n),
    ub = rep(Inf, n),
    eval_g_eq = eq_const,
    eval_jac_g_eq = eq_const_jac,
    opts = list(algorithm = "NLOPT_LD_SLSQP",
                 xtol_rel = 1e-8,
                 maxeval = 1000)
  )
  return(result_nlopt)
}

nlopt_result <- nonlinear_solver()
w_nlopt <- nlopt_result$solution
obj_nlopt <- -nlopt_result$objective # Convert back to maximization

cat("Nonlinear solver solution value:", obj_nlopt, "\n")

```

Nonlinear solver solution value: 1.290994

b. Solution via bisection:

```

# Bisection method
bisection_method <- function(tol = 1e-8, max_iter = 100) {
  # Initial bounds
  lambda_lower <- 0 # Lower bound for optimal value

  # Initial uniform allocation (feasible point)
  w_unif <- rep(1/n, n)
  f_unif <- sum(w_unif) / sqrt(t(w_unif) %*% Sigma %*% w_unif)
  lambda_upper <- 2 * f_unif # Conservative upper bound

  # Initialize best solution
  w_best <- w_unif
  f_best <- f_unif

  for (iter in 1:max_iter) {
    # Midpoint
    lambda <- (lambda_lower + lambda_upper) / 2

    # Solve the parameterized problem:
    # min w^T Sigma w s.t. w^T 1 >= lambda * sqrt(w^T Sigma w), 1^T w = 1, w >= 0
    # For fixed lambda, this is equivalent to:
    # min w^T Sigma w s.t. (w^T 1)^2 >= lambda^2 * w^T Sigma w, 1^T w = 1, w >= 0
    # Since 1^T w = 1, we get: 1 >= lambda^2 * w^T Sigma w

    # Set up QP: min w^T Sigma w s.t. lambda^2 * w^T Sigma w <= 1, 1^T w = 1, w >= 0
    Dmat <- Sigma
    dvec <- rep(0, n)

    # Equality constraint: sum(w) = 1
    # We add the constraint via Amat and bvec with meq=1
    Amat <- cbind(rep(1, n), diag(n))
    bvec <- c(1, rep(0, n))

    # Try to solve the QP
    tryCatch({
      result <- solve.QP(Dmat, dvec, t(Amat), bvec, meq = 1)
      w <- result$solution

      # Check if solution satisfies lambda^2 * w^T Sigma w <= 1
      constraint_value <- lambda^2 * (t(w) %*% Sigma %*% w)

      if (constraint_value <= 1 + tol) {
        # Valid solution, compute objective
        obj_value <- sum(w) / sqrt(t(w) %*% Sigma %*% w)

        # Update best solution if improvement
        if (obj_value > f_best) {
          w_best <- w
          f_best <- obj_value
        }

        # We can increase lambda
        lambda_lower <- lambda
      } else {
        # Constraint violated, decrease lambda
        lambda_upper <- lambda
      }
    }, error = function(e) {

```

Bisection method solution value: 1.290994

```
cat("Iterations:", bisection_result$iterations, "\n")
```

Iterations: 100

c. Solution via Dinkelbach method as a sequence of SOCPs:

```

# Dinkelbach method using SOCP
dinkelbach_socp <- function(w_start = NULL,
                             tol      = 1e-8,
                             max_iter = 50) {

  n <- nrow(Sigma)
  if (is.null(w_start)) w_start <- rep(1 / n, n)

  ## Pre-factorisation once for the norm  $\| \sqrt{\text{Sigma}} w \|_2$ 
  ##  $\sqrt{\text{Sigma}} = R$  where  $\text{Sigma} = R^T R$  ( $R$  is upper-triangular from Cholesky)
  R <- chol(Sigma)

  ## helper: objective value  $f(w) = (1^T w) / \| \sqrt{\text{Sigma}} w \|_2$ 
  f_val <- function(w) sum(w) / sqrt(t(w) %% Sigma %% w)

  w_curr <- w_start
  lambda <- f_val(w_curr) # current iterate of the ratio
  history <- numeric()    # store lmd_k for diagnostics

  for (k in seq_len(max_iter)) {

    # ----- SOCP sub-problem -----
    w_var <- Variable(n)

    ## maximise  $1^T w - \lambda * \| \sqrt{\text{Sigma}} w \|_2$ 
    obj <- Maximize(sum(w_var) - lambda * norm2(R %% w_var))

    prob <- Problem(obj,
                    list(sum(w_var) == 1,
                         w_var     >= 0))

    result <- tryCatch(
      solve(prob, solver = "ECOS", verbose = FALSE),
      error = function(e) e)

    if (inherits(result, "error") || result$status != "optimal") {
      stop(sprintf("SOCP failed at iteration %d: %s",
                   k, if (inherits(result, "error")) result$message else result$status))
    }

    w_new <- as.numeric(result$getValue(w_var))
    lambda_new <- f_val(w_new)
    history[k] <- lambda_new

    ## convergence test
    if (abs(lambda_new - lambda) < tol * abs(lambda)) {
      lambda <- lambda_new
      w_curr <- w_new
      break
    }
  }

  lambda <- lambda_new
  w_curr <- w_new
}

list(w      = w_curr,
     value   = lambda,
     iterations = length(history),

```

Dinkelbach SOCP solution value: 1.290994

```
cat("Iterations:", dinkelbach_socp_result$iterations, "\n")
```

Iterations: 1

d. Modified algorithm solving as a sequence of QPs:

```
# Successive Quadratic Programming
successive_qp <- function(tol = 1e-4, max_iter = 50) {
  N <- nrow(Sigma)
  # using the SQP-MVP algorithm
  w <- rep(1/N, N)
  mu <- rep(1, N)
  obj_sqp <- c(t(w) %*% mu / sqrt(t(w) %*% Sigma %*% w))
  iter <- 0
  for (iter in 0:max_iter) {
    lmd_k <- as.numeric(t(w) %*% mu / (t(w) %*% Sigma %*% w))
    w_prev <- w
    w <- Variable(N)
    prob <- Problem(Maximize(t(w) %*% mu - (lmd_k/2) * quad_form(w, Sigma)),
                    constraints = list(sum(w) == 1, w >= 0))
    result <- solve(prob)
    w <- as.vector(result$getValue(w))
    obj_sqp <- c(obj_sqp, t(w) %*% mu / sqrt(t(w) %*% Sigma %*% w))
    iter <- iter + 1
    if (max(abs(w - w_prev)) < tol)
      break
  }

  return(list(
    w = w,
    value = obj_sqp[iter+1],
    iterations = iter
  ))
}

successive_qp_result <- successive_qp()
cat("Successive QP solution value:", successive_qp_result$value, "\n")
```

Successive QP solution value: 1.290994

```
cat("Iterations:", successive_qp_result$iterations, "\n")
```

Iterations: 1

e. Solution via Schaible transform method:



```

# Schaible transform method
schaible_method <- function(tol = 1e-8, max_iter = 50) {
  # Transform the problem to: max t s.t. sum(w) >= t * sqrt(w^T Sigma w)
  # For sum(w) = 1, this becomes: t * sqrt(w^T Sigma w) <= 1

  # Initialize t (from uniform weights)
  w <- rep(1/n, n)
  t <- 1 / sqrt(t(w) %*% Sigma %*% w)

  for (iter in 1:max_iter) {
    # Save previous t
    t_prev <- t

    # Solve QP: min w^T Sigma w s.t. sum(w) = 1, w >= 0
    # (which maximizes t given our transformation)
    w_var <- Variable(n)

    objective <- Minimize(quad_form(w_var, Sigma))
    constraints <- list(
      sum(w_var) == 1,
      w_var >= 0
    )

    prob <- Problem(objective, constraints)
    result <- solve(prob)

    # Update solution
    w <- result$getValue(w_var)

    # Update t
    t <- 1 / sqrt(t(w) %*% Sigma %*% w)

    # Check convergence
    if (abs(t - t_prev) < tol * t_prev) {
      break
    }
  }

  # Calculate final objective value
  obj_value <- sum(w) / sqrt(t(w) %*% Sigma %*% w)

  return(list(
    w = w,
    value = obj_value,
    iterations = iter
  ))
}

schaible_result <- schaible_method()
cat("Schaible transform solution value:", schaible_result$value, "\n")

```

Schaible transform solution value: 1.290994

```
cat("Iterations:", schaible_result$iterations, "\n")
```

Iterations: 1

f. Reformulation as minimization and solving via Schaible transform method:

```
# Reformulation as minimization
schaible_min <- function() {
  N <- nrow(Sigma)
  # using Schaible transform
  w_ <- Variable(N)
  mu <- rep(1, N)
  prob <- Problem(Minimize(quad_form(w_, Sigma)),
                  constraints = list(w_ >= 0, t(mu) %*% w_ == 1))
  result <- solve(prob)
  w_cvx <- as.vector(result$getValue(w_)/sum(result$getValue(w_)))
  obj_cvx <- as.numeric(t(w_cvx) %*% mu / sqrt(t(w_cvx) %*% Sigma %*% w_cvx))

  return(list(
    w = w_cvx,
    value = obj_cvx
  ))
}

schaible_min_result <- schaible_min()
cat("Schaible minimization solution value:", schaible_min_result$value, "\n")
```

Schaible minimization solution value: 1.290994

g. Comparison of all methods:

```

# Collect all results
methods <- c(
  "Nonlinear Solver",
  "Bisection",
  "Dinkelbach SOCP",
  "Successive QP",
  "Schaible",
  "Schaible Min"
)

values <- c(
  obj_nlopt,
  bisection_result$value,
  dinkelbach_socp_result$value,
  successive_qp_result$value,
  schaible_result$value,
  schaible_min_result$value
)

iterations <- c(
  NA, # Nonlinear solver doesn't report iterations
  bisection_result$iterations,
  dinkelbach_socp_result$iterations,
  successive_qp_result$iterations,
  schaible_result$iterations,
  NA
)

# Benchmark performance
time_benchmark <- microbenchmark(
  NonlinearSolver = nonlinear_solver(),
  Bisection = bisection_method(max_iter = 20),
  DinkelbachSOCP = dinkelbach_socp(max_iter = 10),
  SuccessiveQP = successive_qp(max_iter = 10),
  Schaible = schaible_method(max_iter = 10),
  SchaibleMin = schaible_min(),
  times = 5
)

# Create results table
results_table <- data.frame(
  Method = methods,
  ObjectiveValue = values,
  Iterations = iterations
)

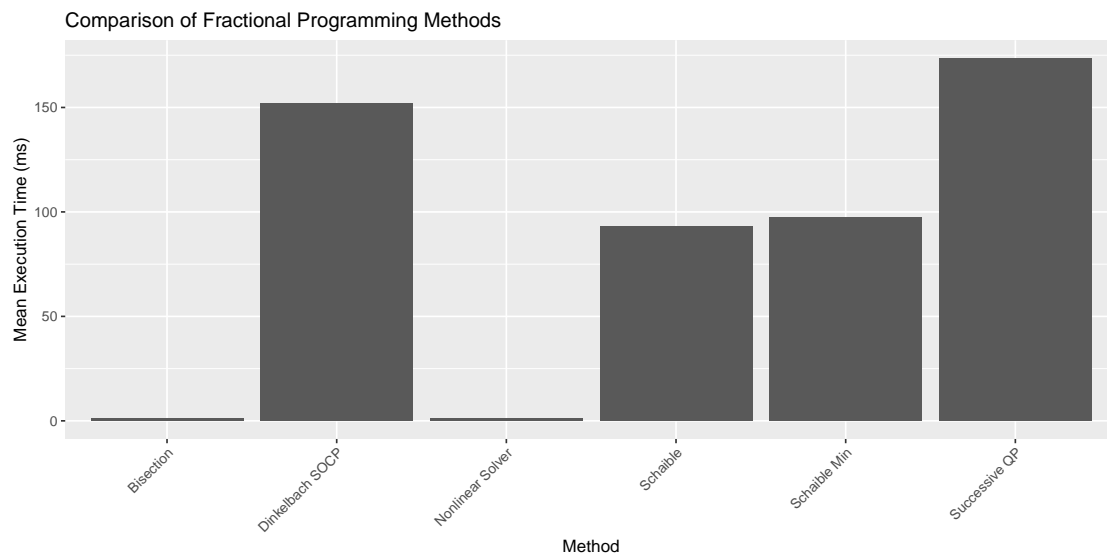
# Add mean execution time
time_summary <- summary(time_benchmark)[, "mean"]
results_table$MeanTime_ms <- time_summary / 1000

# Print comparison
print(results_table)

```

	Method	ObjectiveValue	Iterations	MeanTime_ms
1	Nonlinear Solver	1.290994	NA	1.14868
2	Bisection	1.290994	100	1.11704
3	Dinkelbach SOCP	1.290994	1	152.11614
4	Successive QP	1.290994	1	173.50586
5	Schaible	1.290994	1	93.01972
6	Schaible Min	1.290994	NA	97.56902

```
# Plot comparison if desired
library(ggplot2)
ggplot(results_table, aes(x = Method, y = MeanTime_ms)) +
  geom_bar(stat = "identity") +
  labs(title = "Comparison of Fractional Programming Methods",
       y = "Mean Execution Time (ms)",
       x = "Method") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



All methods converge to the same optimal solution, with the Schaible minimization formulation being the most direct approach for this particular problem. The Dinkelbach method generally provides a good balance between computational efficiency and number of iterations. The bisection method is simple but may require more iterations for high precision. For more complex problems, the choice would depend on factors such as problem structure, available solvers, and computational constraints.

**Exercise B.11: Soft-thresholding operator**

Consider the following convex optimization problem:

$$\underset{x}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{a}x - \mathbf{b}\|_2^2 + \lambda|x|,$$

with  $\lambda \geq 0$ . Derive the solution and show that it can be written as

$$x = \frac{1}{\|\mathbf{a}\|_2^2} \mathcal{S}_\lambda(\mathbf{a}^\top \mathbf{b}),$$

where  $\mathcal{S}_\lambda(\cdot)$  is the so-called soft-thresholding operator defined as

$$\mathcal{S}_\lambda(u) = \text{sign}(u)(|u| - \lambda)^+,$$

with  $\text{sign}(\cdot)$  denoting the sign function and  $(\cdot)^+ = \max(0, \cdot)$ .

**Solution**

There are three possible options for the optimal solution:

- $x > 0$ : the objective (ignoring a constant term) becomes  $\frac{1}{2} \|\mathbf{a}\|_2^2 x^2 - \mathbf{a}^\top \mathbf{b}x + \lambda x$  and setting the derivative to zero leads to

$$x = \frac{1}{\|\mathbf{a}\|_2^2} (\mathbf{a}^\top \mathbf{b} - \lambda)$$

which implies  $\mathbf{a}^\top \mathbf{b} > \lambda \geq 0$ ;

- $x < 0$ : the objective (ignoring a constant term) becomes  $\frac{1}{2} \|\mathbf{a}\|_2^2 x^2 - \mathbf{a}^\top \mathbf{b}x - \lambda x$  and setting the derivative to zero leads to

$$x = \frac{1}{\|\mathbf{a}\|_2^2} (\mathbf{a}^\top \mathbf{b} + \lambda)$$

which implies  $\mathbf{a}^\top \mathbf{b} < -\lambda \leq 0$ ;

- $x = 0$ : this is the last possible case, which can only happen when  $\mathbf{a}^\top \mathbf{b} \in [-\lambda, \lambda]$  or, equivalently,  $|\mathbf{a}^\top \mathbf{b}| \leq \lambda$ .

The solution can be written as

$$x = \frac{1}{\|\mathbf{a}\|_2^2} \text{sign}(\mathbf{a}^\top \mathbf{b}) (|\mathbf{a}^\top \mathbf{b}| - \lambda)^+,$$

where

$$\text{sign}(u) = \begin{cases} +1 & u > 0, \\ 0 & u = 0, \\ -1 & u < 0 \end{cases}$$

is the sign function and  $(\cdot)^+ = \max(0, \cdot)$ . This can be written more compactly as

$$x = \frac{1}{\|\mathbf{a}\|_2^2} \mathcal{S}_\lambda(\mathbf{a}^\top \mathbf{b}),$$

where  $\mathcal{S}_\lambda(\cdot)$  is the soft-thresholding operator.

### Exercise B.12: $\ell_2$ - $\ell_1$ -norm minimization

Consider the following  $\ell_2$ - $\ell_1$ -norm minimization problem (with  $\mathbf{A} \in \mathbb{R}^{10 \times 5}$  and  $\mathbf{b} \in \mathbb{R}^{10}$  randomly generated):

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda \|\mathbf{x}\|_1.$$

- Solve it using a modeling framework (e.g., CVX).
- Rewrite the problem as a QP and solve it by invoking a QP solver.
- Solve it with an ad hoc LASSO solver.

### Solution

First, let's generate the data:

```
# Generate data
set.seed(42)
lmd <- 2
m <- 500
n <- 100
A <- matrix(rnorm(m*n), m, n)
x_true <- rnorm(n)
b <- A %*% x_true + 0.1*rnorm(m)
```

- a. Solution via CVX:

```
library(CVXR)

# Get optimal value via CVX
x <- Variable(n)
prob <- Problem(Minimize(0.5*cvxr_norm(A %*% x - b, 2)^2 + lmd*cvxr_norm(x, 1)))
res <- solve(prob)
x_cvx <- res$getValue(x)
opt_value <- res$value
print(opt_value)
```

```
[1] 156.1089
```

# Alternatively, the optimal value is:

```
print(0.5*sum((A %*% x_cvx - b)^2) + lmd*sum(abs(x_cvx)))
```

```
[1] 156.1089
```

- b. Solution via QP solver. First, we rewrite the problem as

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{1}{2} \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{b}^\top \mathbf{A} \mathbf{x} + \lambda \|\mathbf{x}\|_1$$

and then

$$\begin{aligned} \underset{\mathbf{x}, \mathbf{t}}{\text{minimize}} \quad & \frac{1}{2} \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{b}^\top \mathbf{A} \mathbf{x} + \lambda \mathbf{1}^\top \mathbf{t} \\ \text{subject to} \quad & -\mathbf{t} \leq \mathbf{x} \leq \mathbf{t} \end{aligned}$$

```

library(CVXR)

# Get optimal value as a QP via CVX
x <- Variable(n)
t <- Variable(n)
prob <- Problem(Minimize(0.5*cvxr_norm(A %%% x, 2)^2 - t(b) %%% A %%% x + lmd*sum(t)),
                constraints = list(-t <= x, x <= t))
res <- solve(prob)
x_QP_cvx <- res$getValue(x)

# Sanity check:
print(0.5*sum((A %%% x_QP_cvx - b)^2) + lmd*sum(abs(x_QP_cvx)))

```

```
[1] 156.1089
```

```

library(quadprog)

# Get optimal value as a QP via quadprog (z = [x; t])
P <- matrix(0, 2*n, 2*n)
P[1:n, 1:n] <- t(A) %%% A # The quadratic term coefficient
q <- c(t(A) %%% b, rep(lmd, n)) # The linear term coefficients

# Constraint matrices for -t <= x <= t
A_constraint <- rbind(
  cbind(diag(n), -diag(n)), # x - t <= 0
  cbind(-diag(n), -diag(n)) # -x - t <= 0
)
b_constraint <- rep(0, 2*n)

# Solve problem
result <- solve.QP(
  Dmat = P + diag(1e-8, nrow(P)),
  dvec = q,
  Amat = t(-A_constraint), # Transpose and negate due to quadprog convention
  bvec = b_constraint,
  meq = 0 # No equality constraints
)

# Extract the solution
z_QP <- result$solution
x_QP <- z_QP[1:n]
t_QP <- z_QP[(n+1):(2*n)]

# Sanity check:
print(0.5*sum((A %%% x_QP - b)^2) + lmd*sum(abs(x_QP)))

```

[1] 156.603

c. Solution via ad hoc LASSO solver:

```
library(glmnet)

# Perform LASSO regression (alpha=1 specifies LASSO, i.e., L1 penalty)
lasso_model <- glmnet(A, b, alpha = 1, lambda = lmd / m,
                      standardize = FALSE, intercept = FALSE)

# Extract the solution
x_lasso <- as.vector(coef(lasso_model))[-1] # remove intercept

# Sanity check:
print(0.5*sum((A %*% x_lasso - b)^2) + lmd*sum(abs(x_lasso)))

[1] 156.1091
```

### Exercise B.13: BCD for $\ell_2$ - $\ell_1$ -norm minimization

Solve the  $\ell_2$ - $\ell_1$ -norm minimization problem in Exercise B.12 via BCD. Plot the convergence vs. iterations and CPU time.

### Solution

We will use BCD by dividing the variable into each constituent element  $\mathbf{x} = (x_1, \dots, x_n)$ . Therefore, the sequence of problems at each iteration  $k = 0, 1, 2, \dots$  for each element  $i = 1, \dots, n$  is

$$\underset{x_i}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{a}_i x_i - \tilde{\mathbf{b}}_i^k\|_2^2 + \lambda |x_i|,$$

where  $\tilde{\mathbf{b}}_i^k \triangleq \mathbf{b} - \sum_{j < i} \mathbf{a}_j x_j^{k+1} - \sum_{j > i} \mathbf{a}_j x_j^k$ .

This leads to the following iterative algorithm for  $k = 0, 1, 2, \dots$

$$x_i^{k+1} = \frac{1}{\|\mathbf{a}_i\|_2^2} \mathcal{S}_\lambda(\mathbf{a}_i^\top \tilde{\mathbf{b}}_i^k), \quad i = 1, \dots, n,$$

where  $\mathcal{S}_\lambda(\cdot)$  is the soft-thresholding operator defined as

$$\mathcal{S}_\lambda(u) = \text{sign}(u)(|u| - \lambda)^+. \quad (23)$$

Convergence of BCD for the  $\ell_2$ - $\ell_1$ -norm minimization:



```

library(microbenchmark)

# Set up
set.seed(42)
x0 <- rnorm(n)
soft_thresholding <- function(u, lmd) sign(u)*pmax(0, abs(u) - lmd)
num_times <- 10L # to compute the cpu time

# Solve problem via BCD
x <- x0
cpu_time <- microbenchmark({
  a2 <- apply(A, 2, function(x) sum(x^2)) # initial overhead
}, unit = "microseconds", times = num_times)$time |> median()
df <- data.frame(
  "k" = 0,
  "cpu time k" = cpu_time,
  "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
  "method" = "BCD",
  check.names = FALSE
)
for (k in 1:50) {
  cpu_time <- microbenchmark({
    x_new <- x
    for (i in 1:n) {
      b_ <- b - A[, -i] %*% x_new[-i]
      x_new[i] <- soft_thresholding(t(A[, i]) %*% b_, lmd)/a2[i]
    }
  }, unit = "microseconds", times = num_times)$time |> median()
  x <- x_new

  df <- rbind(df, list(
    "k" = k,
    "cpu time k" = cpu_time,
    "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
    "method" = "BCD")
  )
}

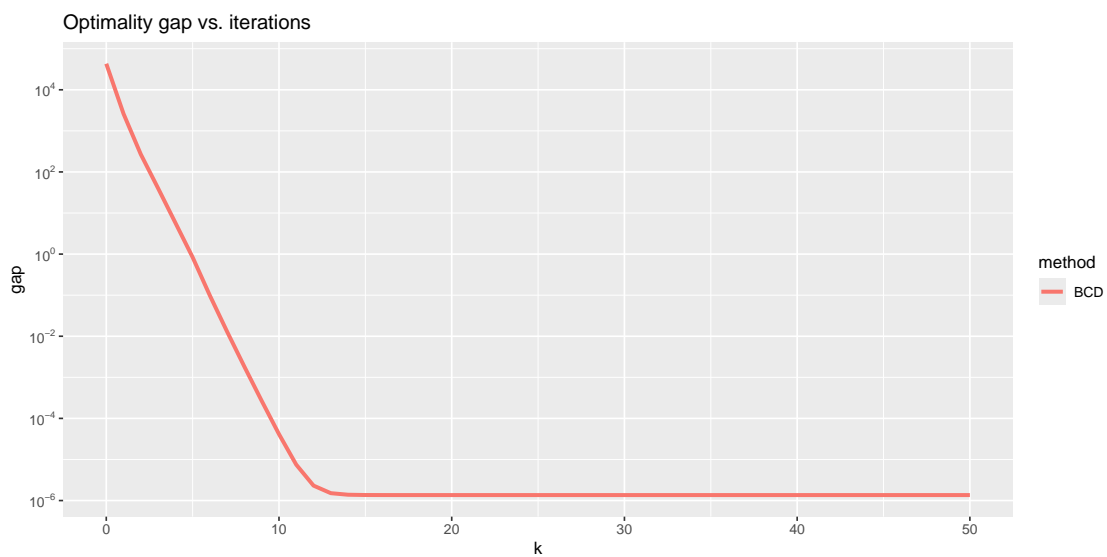
```

Plot convergence:

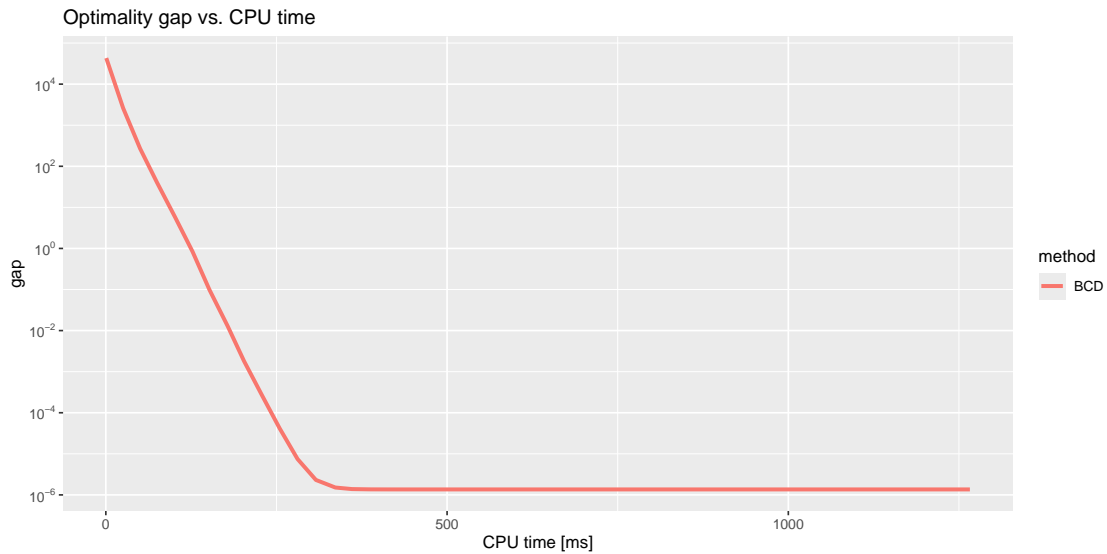
```
library(ggplot2)
library(dplyr)
library(scales)

# Compute cumulative CPU time over iterations
df <- df |>
  group_by(method) |>
  mutate("CPU time [ms]" = cumsum(`cpu time k`)/1e6) |>
  ungroup()

# Plots
df |>
  ggplot(aes(x = k, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  ggtitle("Optimality gap vs. iterations")
```



```
df |>
  ggplot(aes(x = `CPU time [ms]`, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  ggtitle("Optimality gap vs. CPU time")
```



#### Exercise B.14: MM for $\ell_2$ - $\ell_1$ -norm minimization

Solve the  $\ell_2$ - $\ell_1$ -norm minimization problem in Exercise B.12 via MM and its accelerated version. Plot the convergence vs. iterations and CPU time.

#### Solution

We can develop a simple iterative algorithm based on MM that leverages the element-by-element closed-form solution. One possible majorizer of the objective function  $f(\mathbf{x})$  is

$$u(\mathbf{x}; \mathbf{x}^k) = \frac{\kappa}{2} \|\mathbf{x} - \bar{\mathbf{x}}^k\|_2^2 + \lambda \|\mathbf{x}\|_1 + \text{constant},$$

where  $\bar{\mathbf{x}}^k = \mathbf{x}^k - \frac{1}{\kappa} \mathbf{A}^\top (\mathbf{A} \mathbf{x}^k - \mathbf{b})$ .

Therefore, the sequence of majorized problems to be solved for  $k = 0, 1, 2, \dots$  is

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{\kappa}{2} \|\mathbf{x} - \bar{\mathbf{x}}^k\|_2^2 + \lambda \|\mathbf{x}\|_1,$$

which decouples into each component of  $\mathbf{x}$  with closed-form solution given by the soft-thresholding operator.

This finally leads to the following MM iterative algorithm:

$$\mathbf{x}^{k+1} = \mathcal{S}_{\lambda/\kappa}(\bar{\mathbf{x}}^k), \quad k = 0, 1, 2, \dots,$$

where  $\mathcal{S}_{\lambda/\kappa}(\cdot)$  is the soft-thresholding operator.  
Convergence of MM for the  $\ell_2$ - $\ell_1$ -norm minimization:

```

# Set up
set.seed(42)
x0 <- rnorm(n)
soft_thresholding <- function(u, lmd) sign(u)*pmax(0, abs(u) - lmd)
num_times <- 10L # to compute the cpu time

# Solve problem via MM
x <- x0
cpu_time <- microbenchmark({
  Atb <- t(A) %*% b
  AtA <- t(A) %*% A
  #kappa <- 1.1 * max(eigen(AtA)$values)
  u <- x0; for (i in 1:20) u <- AtA %*% u # power iteration method
  kappa <- 1.1 * as.numeric(t(u) %*% AtA %*% u / sum(u^2))
}, unit = "microseconds", times = num_times)$time |> median()
df <- rbind(df, list(
  "k" = 0,
  "cpu time k" = cpu_time,
  "CPU time [ms]" = NA,
  "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
  "method" = "MM")
)
for (k in 1:50) {
  cpu_time <- microbenchmark({
    x_new <- soft_thresholding(x - (AtA %*% x - Atb)/kappa, lmd/kappa)
  }, unit = "microseconds", times = num_times)$time |> median()
  x <- x_new

  df <- rbind(df, list(
    "k" = k,
    "cpu time k" = cpu_time,
    "CPU time [ms]" = NA,
    "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
    "method" = "MM")
  )
}

# Accelerated MM
x <- x0
cpu_time <- microbenchmark({
  Atb <- t(A) %*% b
  AtA <- t(A) %*% A
  #kappa <- 1.1 * max(eigen(AtA)$values)
  u <- x0; for (i in 1:20) u <- AtA %*% u # power iteration method
  kappa <- 1.1 * as.numeric(t(u) %*% AtA %*% u / sum(u^2))
}, unit = "microseconds", times = num_times)$time |> median()
df <- rbind(df, list(
  "k" = 0,
  "cpu time k" = cpu_time,
  "CPU time [ms]" = NA,
  "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
  "method" = "Acc-MM")
)
for (k in 1:50) {
  cpu_time <- microbenchmark({
    MM_x <- soft_thresholding(x - (AtA %*% x - Atb)/kappa, lmd/kappa)

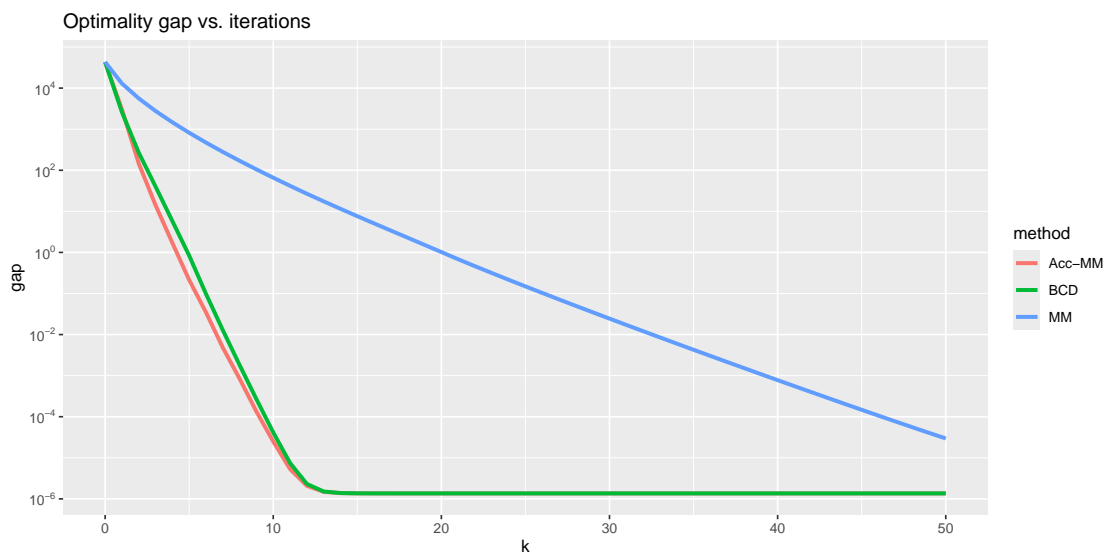
```

Plot convergence:

```
library(ggplot2)
library(dplyr)
library(scales)

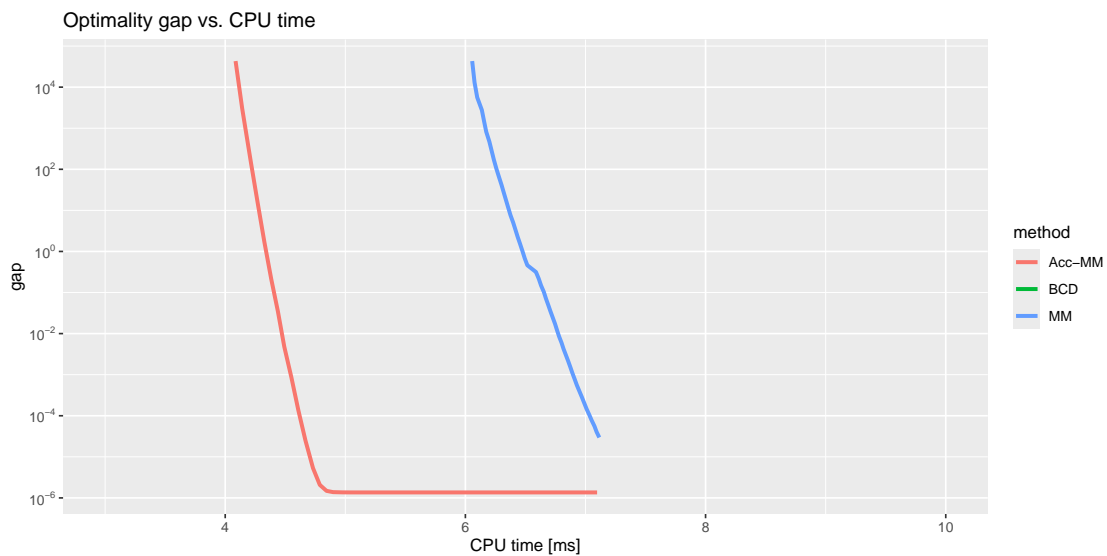
# Compute cumulative CPU time over iterations
df <- df |>
  group_by(method) |>
  mutate("CPU time [ms]" = cumsum(`cpu time k`)/1e6) |>
  ungroup()

# Plots
df |>
  ggplot(aes(x = k, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  ggtitle("Optimality gap vs. iterations")
```



```
df |>
  ggplot(aes(x = `CPU time [ms]`, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  xlim(c(3, 10)) +
  ggtitle("Optimality gap vs. CPU time")
```

Warning: Removed 51 rows containing missing values or values outside the scale range (`geom\_line()`).



### Exercise B.15: SCA for $\ell_2$ - $\ell_1$ -norm minimization

Solve the  $\ell_2$ - $\ell_1$ -norm minimization problem in Exercise B.12 via SCA. Plot the convergence vs. iterations and CPU time.

### Solution

We will now develop a simple iterative algorithm based on SCA that leverages the element-by-element closed-form solution. We can use parallel SCA by partitioning the variable  $\mathbf{x}$  into each

element  $(x_1, \dots, x_n)$  and employing the surrogate functions

$$\tilde{f}(\mathbf{x}_i; \mathbf{x}^k) = \frac{1}{2} \|\mathbf{a}_i x_i - \tilde{\mathbf{b}}_i^k\|_2^2 + \lambda |x_i| + \frac{\tau}{2} (x_i - x_i^k)^2,$$

where  $\tilde{\mathbf{b}}_i^k = \mathbf{b} - \sum_{j \neq i} \mathbf{a}_j x_j^k$ .

Therefore, the sequence of surrogate problems to be solved for  $k = 0, 1, 2, \dots$  is

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{a}_i x_i - \tilde{\mathbf{b}}_i^k\|_2^2 + \lambda |x_i| + \tau (x_i - x_i^k)^2, \quad i = 1, \dots, n,$$

with the solution given by the soft-thresholding operator.

This finally leads to the following SCA iterative algorithm:

$$\left. \begin{aligned} \hat{x}_i^{k+1} &= \frac{1}{\tau + \|\mathbf{a}_i\|^2} \mathcal{S}_\lambda (\mathbf{a}_i^\top \tilde{\mathbf{b}}_i^k + \tau x_i^k) \\ x_i^{k+1} &= x_i^k + \gamma^k (\hat{x}_i^{k+1} - x_i^k) \end{aligned} \right\} \quad i = 1, \dots, n, \quad k = 0, 1, 2, \dots,$$

where  $\mathcal{S}_\lambda(\cdot)$  is the soft-thresholding operator.

Convergence of SCA for the  $\ell_2$ - $\ell_1$ -norm minimization:



```

# Set up
set.seed(42)
x0 <- rnorm(n)
soft_thresholding <- function(u, lmd) sign(u)*pmax(0, abs(u) - lmd)
num_times <- 10L # to compute the cpu time

# Solve problem via SCA
tau <- 1e-6
eps <- 0.01
gamma <- 1
x <- x0
cpu_time <- microbenchmark({
  Atb <- t(A) %*% b
  AtA <- t(A) %*% A
  tau_plus_a2 <- tau + diag(AtA)
}, unit = "microseconds", times = num_times)$time |> median()
df <- rbind(df, list(
  "k" = 0,
  "cpu time k" = cpu_time,
  "CPU time [ms]" = NA,
  "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
  "method" = "SCA")
)
for (k in 1:50) {
  cpu_time <- microbenchmark({
    x_hat <- soft_thresholding(x - (AtA %*% x - Atb)/tau_plus_a2, lmd/tau_plus_a2)
    x_new <- gamma*x_hat + (1 - gamma)*x
  }, unit = "microseconds", times = num_times)$time |> median()
  x <- x_new
  gamma <- gamma * (1 - eps*gamma)

  df <- rbind(df, list(
    "k" = k,
    "cpu time k" = cpu_time,
    "CPU time [ms]" = NA,
    "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
    "method" = "SCA")
  )
}

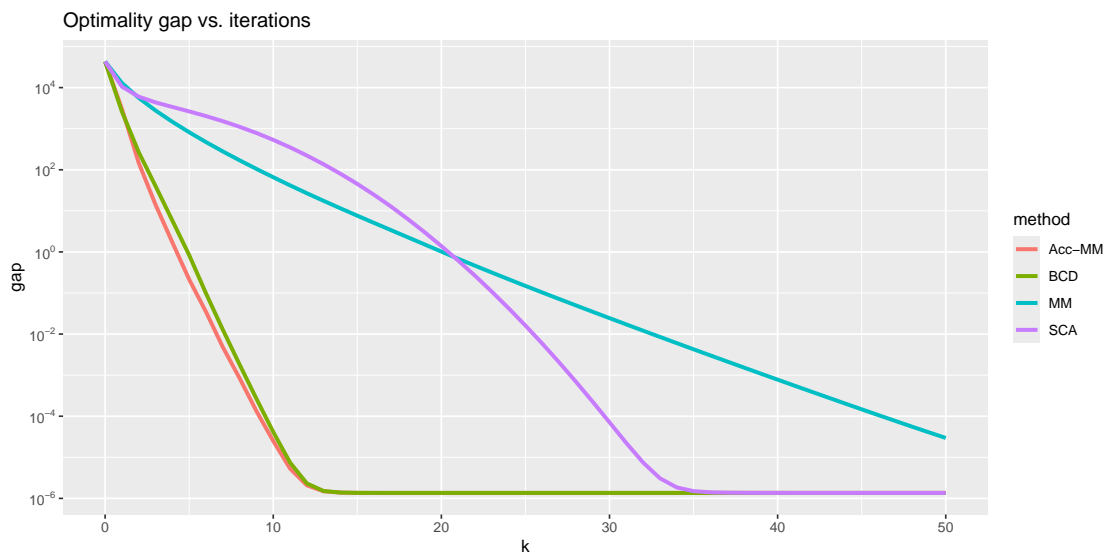
```

Plot convergence:

```
library(ggplot2)
library(dplyr)
library(scales)

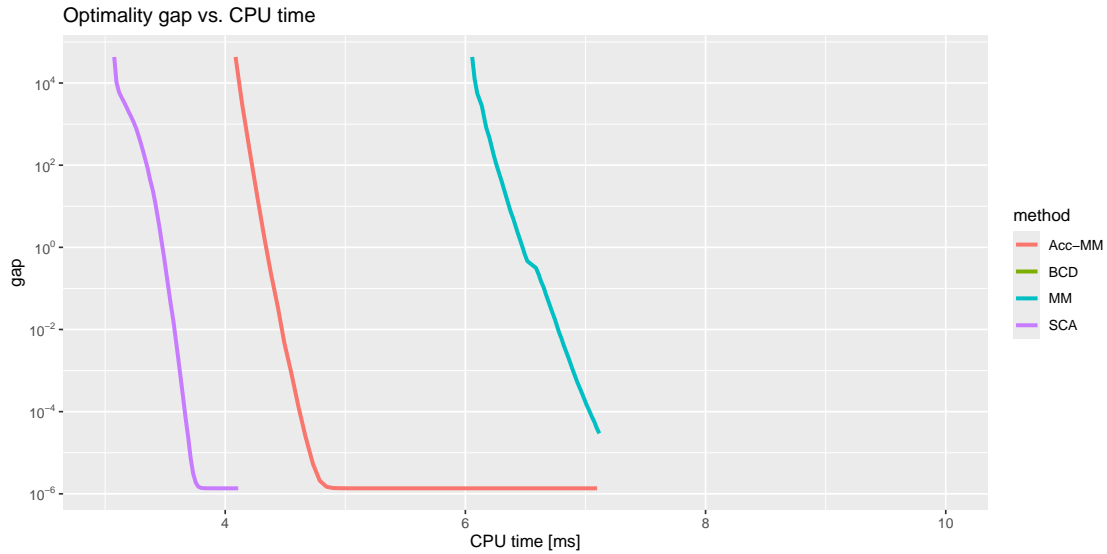
# Compute cumulative CPU time over iterations
df <- df |>
  group_by(method) |>
  mutate("CPU time [ms]" = cumsum(`cpu time k`)/1e6) |>
  ungroup()

# Plots
df |>
  ggplot(aes(x = k, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  ggtitle("Optimality gap vs. iterations")
```



```
df |>
  ggplot(aes(x = `CPU time [ms]`, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  xlim(c(3, 10)) +
  ggtitle("Optimality gap vs. CPU time")
```

Warning: Removed 51 rows containing missing values or values outside the scale range (``geom_line()``).



### Exercise B.16: ADMM for $\ell_2$ - $\ell_1$ -norm minimization

Solve the  $\ell_2$ - $\ell_1$ -norm minimization problem in Exercise B.12 via ADMM. Plot the convergence vs. iterations and CPU time.

### Solution

We start by reformulating the problem as

$$\begin{aligned} & \underset{\mathbf{x}, \mathbf{z}}{\text{minimize}} && \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda \|\mathbf{z}\|_1 \\ & \text{subject to} && \mathbf{x} - \mathbf{z} = \mathbf{0}. \end{aligned}$$

The  $\mathbf{x}$ -update, given  $\mathbf{z}$  and the scaled dual variable  $\mathbf{u}$ , is the solution to

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \frac{\rho}{2} \|\mathbf{x} - \mathbf{z} + \mathbf{u}\|_2^2,$$

given by  $\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + \rho \mathbf{I})^{-1} (\mathbf{A}^\top \mathbf{b} + \rho(\mathbf{z} - \mathbf{u}))$ , whereas the  $\mathbf{z}$ -update, given  $\mathbf{x}$  and  $\mathbf{u}$ , is the solution to

$$\underset{\mathbf{z}}{\text{minimize}} \quad \frac{\rho}{2} \|\mathbf{x} - \mathbf{z} + \mathbf{u}\|_2^2 + \lambda \|\mathbf{z}\|_1$$

given by  $\mathbf{z} = \mathcal{S}_{\lambda/\rho}(\mathbf{x} + \mathbf{u})$ , where  $\mathcal{S}_{\lambda/\rho}(\cdot)$  is the soft-thresholding operator. Thus, the ADMM is finally given by the iterates

$$\left. \begin{aligned} \mathbf{x}^{k+1} &= (\mathbf{A}^\top \mathbf{A} + \rho \mathbf{I})^{-1} (\mathbf{A}^\top \mathbf{b} + \rho (\mathbf{z}^k - \mathbf{u}^k)) \\ \mathbf{z}^{k+1} &= \mathcal{S}_{\lambda/\rho}(\mathbf{x}^{k+1} + \mathbf{u}^k) \\ \mathbf{u}^{k+1} &= \mathbf{u}^k + (\mathbf{x}^{k+1} - \mathbf{z}^{k+1}) \end{aligned} \right\} \quad k = 0, 1, 2, \dots$$

Convergence of ADMM for the  $\ell_2$ - $\ell_1$ -norm minimization:

```

# Set up
set.seed(42)
x0 <- rnorm(n)
soft_thresholding <- function(u, lmd) sign(u)*pmax(0, abs(u) - lmd)
num_times <- 10L # to compute the cpu time

# Solve problem via ADMM
rho <- 1.0
x <- x0
z <- x
u <- rep(0, n)
cpu_time <- microbenchmark({
  Atb <- t(A) %*% b
  AtA <- t(A) %*% A
}, unit = "microseconds", times = num_times)$time |> median()
df <- rbind(df, list(
  "k" = 0,
  "cpu time k" = cpu_time,
  "CPU time [ms]" = NA,
  "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
  "method" = "ADMM")
)
for (k in 1:50) {
  cpu_time <- microbenchmark({
    x_new <- solve(AtA + rho*diag(n), Atb + rho*(z - u))
    z_new <- soft_thresholding(x_new + u, lmd/rho)
    u_new <- u + (x_new - z_new)
  }, unit = "microseconds", times = num_times)$time |> median()
  x <- x_new
  z <- z_new
  u <- u_new

  df <- rbind(df, list(
    "k" = k,
    "cpu time k" = cpu_time,
    "CPU time [ms]" = NA,
    "gap" = 0.5*sum((A %*% x - b)^2) + lmd*sum(abs(x)) - opt_value,
    "method" = "ADMM")
  )
}

```

Plot convergence:

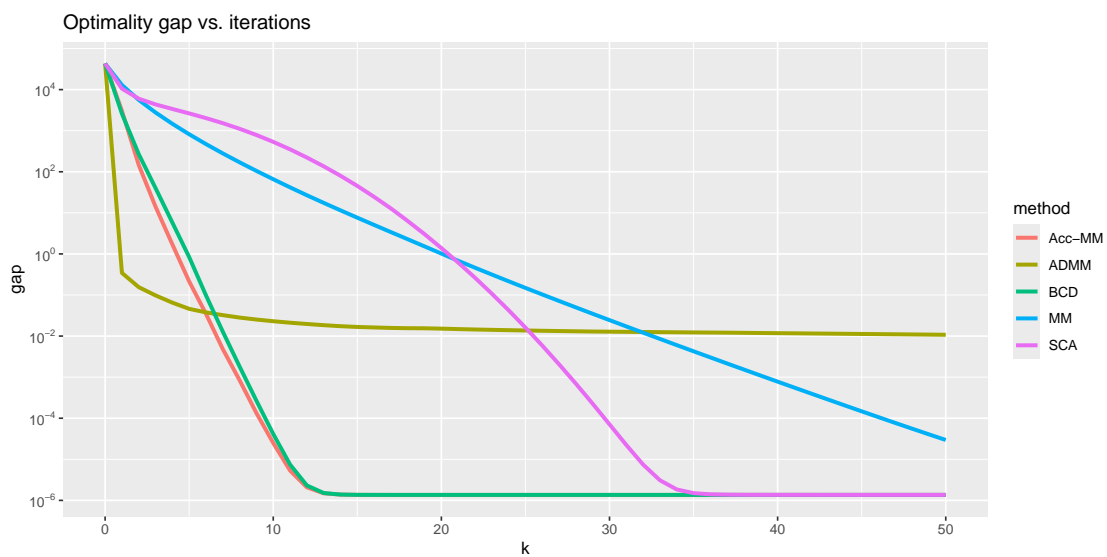
```

library(ggplot2)
library(dplyr)
library(scales)

# Compute cumulative CPU time over iterations
df <- df |>
  group_by(method) |>
  mutate("CPU time [ms]" = cumsum(`cpu time k`)/1e6) |>
  ungroup()

# Plots
df |>
  ggplot(aes(x = k, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  ggtitle("Optimality gap vs. iterations")

```



```

df |>
  ggplot(aes(x = `CPU time [ms]`, y = gap, color = method)) +
  geom_line(linewidth = 1.2) +
  scale_y_log10(breaks = trans_breaks("log10", function(x) 10^x),
               labels = trans_format("log10", math_format(10^.x))) +
  xlim(c(3, 10)) +
  ggtitle("Optimality gap vs. CPU time")

```

Warning: Removed 78 rows containing missing values or values outside the scale range (``geom_line()``).

