Comparing Unconstrained Optimization Methods

Henrique Ap. Laureano
ID 158811
KAUST

Spring Semester
2018

In this problem, we will solve the linear least squares (LLS) problem below:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \| \mathbf{Ax} - \mathbf{b} \|_2^2$$

The matrix $\mathbf{A}$ and vector $\mathbf{b}$ are saved in *Ab.mat* provided with the homework. In what follows, you will implement your own version of the different unconstrained optimization methods we talked about in class. Submit all your code.

Here I'm doing everything in R.

```
# <r code> =============================================================== #
path <- "~/Dropbox/KAUST/numerical_optimization/hw3/"                # files path

library(R.matlab)                            # loading library to read the datasets
                                         # reading A, the function import as a list class
data_a <- readMat(paste0(path, "data_a.mat"))

a <- matrix(unlist(data_a), 1e4, 1e3)                        # converting to a matrix

data_b <- readMat(paste0(path, "data_b.mat"))                              # reading b

b <- matrix(unlist(data_b), 1e4, 1)                          # converting to a matrix
# </r code> ============================================================== #
```

(a)

Use the stationarity equation to compute $\mathbf{x}^*$. What is $\| \mathbf{Ax}^* - \mathbf{b} \|_2$? Based on this result, is $\mathbf{b} \in \mathcal{R}(A)$?

Solution:

$$f(\mathbf{x}) = \| \mathbf{A}\mathbf{x} - \mathbf{b} \|_2^2 = (\mathbf{A}\mathbf{x} - \mathbf{b})^\top (\mathbf{A}\mathbf{x} - \mathbf{b})$$
$$= \mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x} - 2\mathbf{x}^\top \mathbf{A}^\top \mathbf{b} + \mathbf{b}^\top \mathbf{b}$$
$$\triangledown f(\mathbf{x}) = 2\mathbf{A}^\top \mathbf{A}\mathbf{x} - 2\mathbf{A}^\top \mathbf{b}$$

$$\triangledown f(\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad 2\mathbf{A}^\top \mathbf{A}\mathbf{x} = 2\mathbf{A}^\top \mathbf{b}$$
$$\mathbf{x}^* = (\mathbf{A}^\top \mathbf{A}\mathbf{x})^{-1} \mathbf{A}^\top \mathbf{b}$$

```r
# <r code> =================================================================== #
x.ast <- solve(t(a) %*% a) %*% t(a) %*% b          # computing \mathbf{x}^{\ast}

norm(a %*% x.ast - b, type = "2")                          # computing the L2 norm
# </r code> ================================================================== #
```

```
[1] 31.66644
```

Is $\mathbf{b} \in \mathcal{R}(A)$? No, because $\| \mathbf{A}\mathbf{x}^* - \mathbf{b} \|_2 = 31.66 > 0$.

□

(b)

Implement steepest descent with exact line search and apply it to the LLS problem above. Initialize at $\mathbf{x}_0 = \mathbf{0}$ and stop your descent loop at iteration k when $\| \mathbf{x}^* - \mathbf{x}_k \|_2 \leq \| \mathbf{x}^* \|_2 * 10^{-6}$. Plot the evolution of the objective value (in log scale) and plot $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k. Discuss your findings.

Solution:

Steepest descent iteration $k+1$ with exact line search:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\triangledown f_k^\top \triangledown f_k}{\triangledown f_k^\top 2\mathbf{A}^\top \mathbf{A} \triangledown f_k} \triangledown f_k,$$

with $2\mathbf{A}^\top \mathbf{A}$ being the Hessian of $f(\mathbf{x})$ and with the step length $\alpha_k = \frac{\triangledown f_k^\top \triangledown f_k}{\triangledown f_k^\top 2\mathbf{A}^\top \mathbf{A} \triangledown f_k}$.

```r
# <r code> =================================================================== #
steep.desc_els <- function(x, a, b) {    # steepest descent with exact line search
  x_k = matrix(rep(0, length(x)))    # initializing at \mathbf{x}_{0} = \mathbf{0}
  i = 1                                               # setting iterations counter
                                                         # defining stop criterium
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
    grad = 2 * t(a) %*% (a %*% x_k[ , i] - b)                # computing gradient
    alpha = (t(grad) %*% grad) / (2 * t(grad) %*% t(a) %*% a %*% grad)     # \alpha
    x_k = cbind(x_k, x_k[ , i] - as.numeric(alpha) * grad)       # putting together
    i = i + 1                                                   # addying new iteration
  }
  return(xks = x_k[ , -1])      # returning the \mathbf{x}_{k}'s of each iteration
}
xks <- steep.desc_els(x.ast, a, b)                              # running the function

iter <- ncol(xks)                                               # number of iterations
                      # empty object to keep the objective value of each iteration
obj.value <- numeric(iter)
                                 # computing the objective value of each iteration
for (i in 1:iter) obj.value[i] = norm(a %*% xks[ , i] - b, type = "2")**2

norms <- numeric(iter)  # empty object to keep the L2 norm value of each iteration
                                         # computing the L2 norm of each iteration
for(i in 1:iter) norms[i] = norm(x.ast - xks[ , i], type = "2")

par(mfrow = c(1, 2), mar = c(4, 4, 3, 2) + .1)                 # graphical definitions
plot(obj.value, log = "y", type = "b"
     , xlab = "Iteration", ylab = "Objective value (in log scale)", main = "(a)")
plot(norms, type = "b", xlab = "Iteration", ylab = "L2 norm", main = "(b)")
# </r code> ==================================================================== #
```

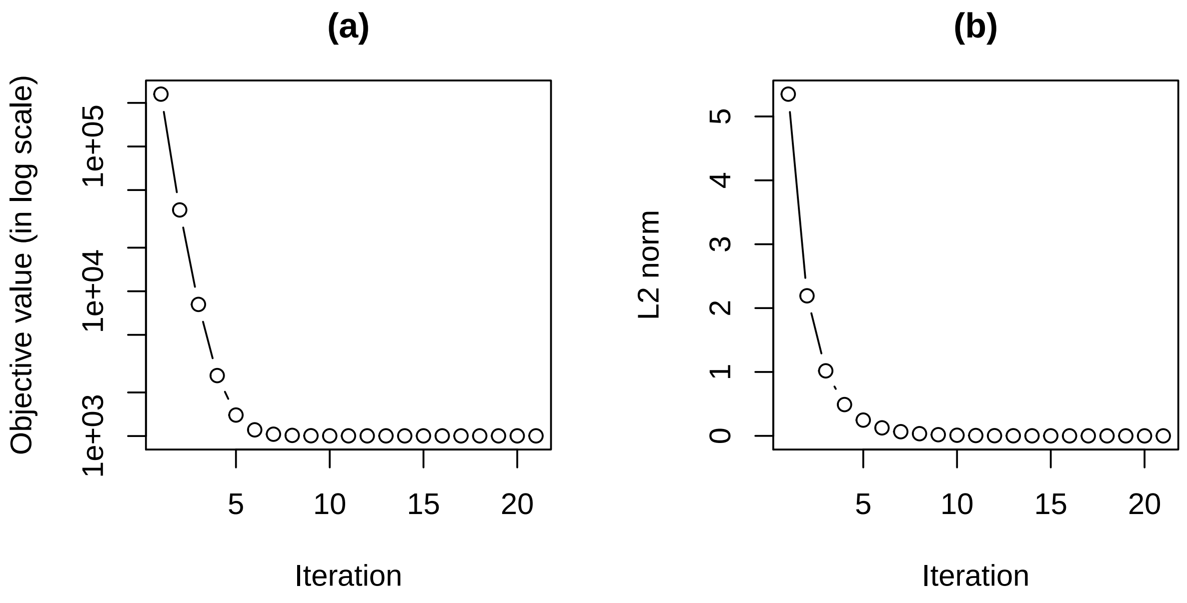


Figure 1: Steepest descent with exact line search. (a): The objective value (in log scale) with increasing k; (b): $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k.

We see that the objective value and $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ decrease quickly as k increase.

```r
# <r code> =================================================================== #
obj.value[iter]                                            # final objective value
norms[iter]                                                           # final norm
# </r code> ================================================================== #
```

```
[1] 1002.763
[1] 1.248952e-05
```

□

(c)

Implement steepest descent with backtracking. Use the same initialization and stopping criterion as in (b). Plot the evolution of the objective value (in log scale) and plot $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k. Compare these results to those of **(b)** and discuss.

Solution:

Steepest descent iteration $k+1$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \frac{\bigtriangledown f_k}{\| \bigtriangledown f_k \|_2}, \quad \text{with the step length } \alpha_k \text{ gave by the backtracking line search.}$$

Algorithm 3.1 (Backtracking Line Search).

```
Choose $\bar{\alpha} > 0$, $\rho \in (0, 1)$, $c \in (0, 1)$; Set $\alpha \leftarrow \bar{\alpha}$;
repeat until $f(x_k + \alpha p_k) \leq f(x_k) + c\alpha \nabla f_k^T p_k$
    $\alpha \leftarrow \rho\alpha$;
end (repeat)
Terminate with $\alpha_k = \alpha$.
```

```r
# <r code> =================================================================== #
steep.desc_back <- function (x, a, b) {        # steepest descent with backtracking
  x_k = matrix(rep(0, length(x)))    # initializing at \mathbf{x}_{0} = \mathbf{0}
  i = 1                                                # setting iterations counter
  fn <- function(x, a, b) {                                  # function to compute:
    obj = norm(a %*% x - b, type = "2")**2                           # object value
    grad = 2 * t(a) %*% (a %*% x - b)                                    # gradient
    return(list(obj = obj, grad = grad))
  }                   # function to compute the search direction (steepest descent)
  steep <- function(fn) with(fn, - grad / norm(grad, type = "2"))
  alpha = 1                                            # initializing at \alpha = 1
           # setting \rho and the constant as 0.7 (to be used in the backtracking)
  rho = const = .7
                       # empthy object to keep the object values at each iteration
  obj.value = numeric(1)
```

```
                              # empty object to count how many times the algorithm
  count = numeric(1)          #        enters in the backtracking at each iteration
  norms = numeric(1)            # empty object to keep the norms at each iteration
                   # computing the object value and the gradient for the first time
  fn_k = fn(x_k[ , i], a, b)
  direc = steep(fn_k)                                # computing the search direction
                              # initializing while loop with the stopping criterion
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
    while (fn(x_k[ , i] + alpha * direc, a, b)$obj >   # initializing backtracking
           fn_k$obj + const * alpha * t(fn_k$grad) %*% direc) {
                 # counting how many times we do the backtracking at each iteration
      count[i] = count[i] + 1
      alpha = rho * alpha                 # updating \alpha (doing the backtracking)
    }
    x_k = cbind(x_k, x_k[ , i] + alpha * direc) # computing the new \mathbf{x}_{k}
    i = i + 1                                           # updating iteration counter
    fn_k = fn(x_k[ , i], a, b)           # computing new object value and gradient
    obj.value[i] = fn_k$obj                            # keeping the new object value
    norms[i] = norm(x - x_k[ , i], type = "2")     # computing and keeping the norm
    direc = steep(fn_k)                            # computing the new search direction
    count[i] = 0                    # setting the count to zero for the new iteration
  }
    # returning the \mathbf{x}_{k}'s, object values, norms and backtracking counts
  return(list(x = x_k[ , -1], obj.value = obj.value[-1], norms = norms[-1]
              , count = count))
}                                 # running the steepest descent with backtracking
xks <- steep.desc_back(x.ast, a, b)

tail(xks$obj.value, 1)                                      # final objective value
tail(xks$norms, 1)                                                     # final norm
xks$count                 # number of times that we do backtracking at each iteration
# </r code> ============================================================ #

[1] 1002.763
[1] 1.474028e-05
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 3 2 1 2 2 2 3 1 2 2 3 2 0
```

Comparing with **(b)** - steepest descent with exact line search, we have the same final results (values). However, with steepest descent with backtracking we see more iterations with the values decreasing more slowly.

```
# <r code> ============================================================= #
par(mfrow = c(1, 2), mar = c(4, 4, 3, 2) + .1)            # graphical definitions
plot(xks$obj.value, log = "y", type = "b" # plotting the object values (log scale)
     , xlab = "Iteration", ylab = "Objective value (in log scale)", main = "(a)")
                                       # plotting norms with increasing iterations k
plot(xks$norms, type = "b", xlab = "Iteration", ylab = "L2 norm", main = "(b)")
# </r code> ============================================================ #
```

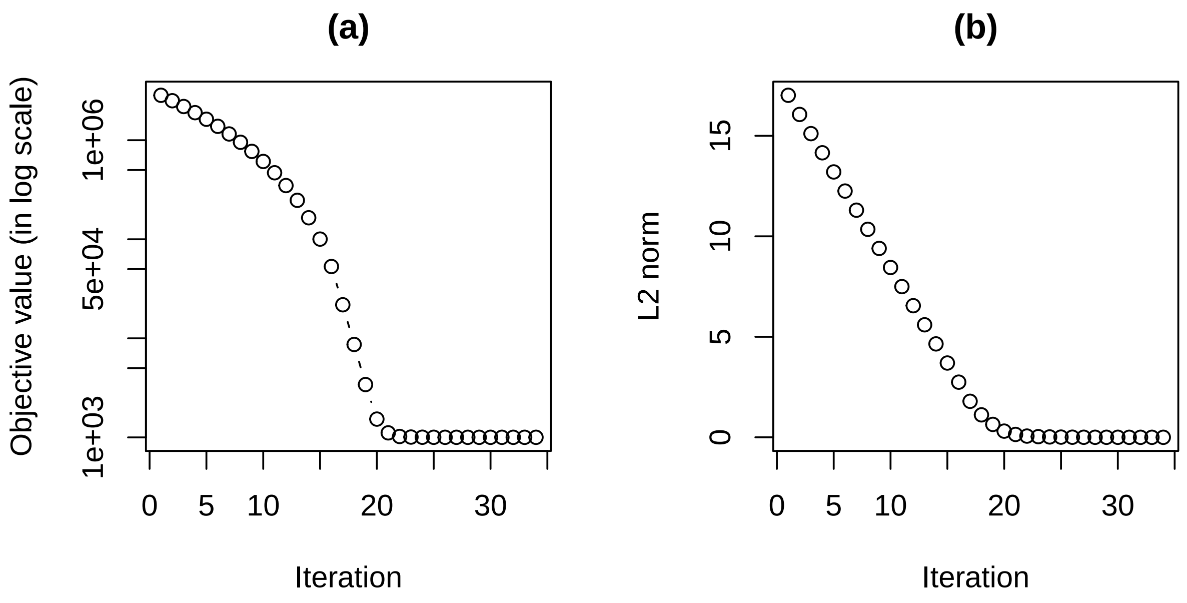


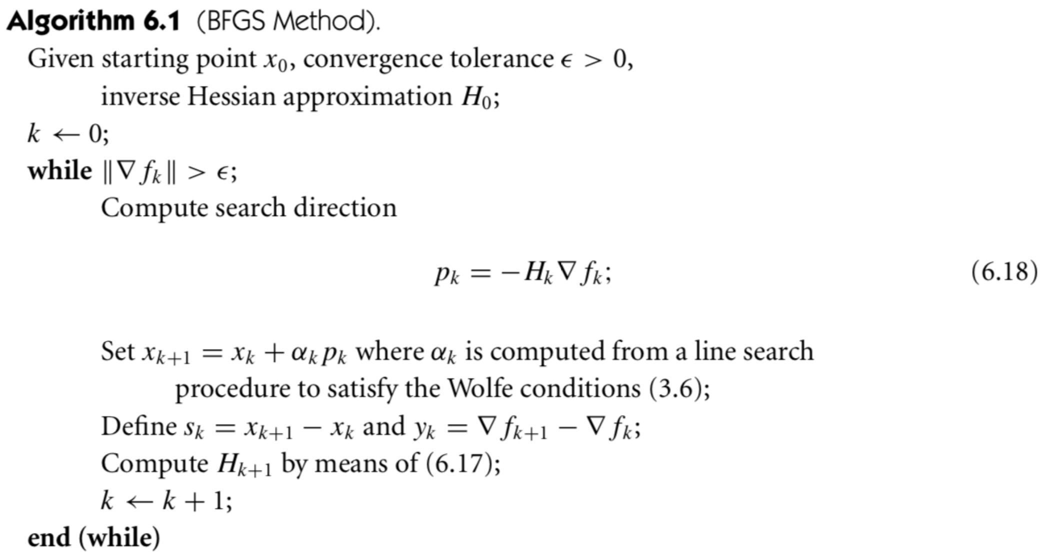
Figure 2: Steepest descent with backtracking. (a): The objective value (in log scale) with increasing k; (b): $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k.

□

(d)

Implement BFGS using backtracking. Use the same initialization and stopping criterion as in (b). Plot the evolution of the objective value (in log scale) and plot $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k. Compare these results to **(c)** and discuss.

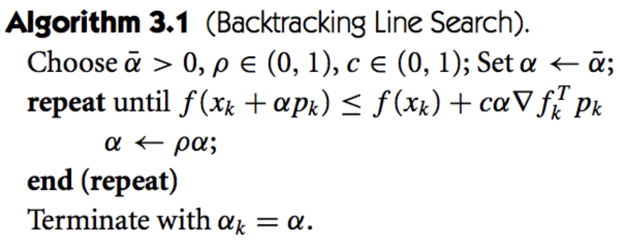
Solution:



```
Algorithm 6.1 (BFGS Method).
 Given starting point $x_0$, convergence tolerance $\epsilon > 0$,
        inverse Hessian approximation $H_0$;
 $k \leftarrow 0$;
 while $\|\nabla f_k\| > \epsilon$;
        Compute search direction
                    $p_k = -H_k \nabla f_k;$          (6.18)
        Set $x_{k+1} = x_k + \alpha_k p_k$ where $\alpha_k$ is computed from a line search
               procedure to satisfy the Wolfe conditions (3.6);
        Define $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$;
        Compute $H_{k+1}$ by means of (6.17);
        $k \leftarrow k + 1$;
 end (while)
```

with $\quad \mathbf{H}_{k+1} = (\mathbf{I} - \boldsymbol{\rho}_k \mathbf{s}_k \mathbf{y}_k^\top)\mathbf{H}_k(\mathbf{I} - \boldsymbol{\rho}_k \mathbf{y}_k \mathbf{s}_k^\top) + \boldsymbol{\rho}_k \mathbf{s}_k \mathbf{s}_k^\top, \quad$ and $\boldsymbol{\rho}_k$ defined by $\boldsymbol{\rho}_k = \dfrac{1}{\mathbf{y}_k^\top \mathbf{s}_k}$.

Here the stopping criterion is different (see problem statement) and the step length α_k is gave by the backtracking line search.



Algorithm 3.1 (Backtracking Line Search).

```
Choose $\bar{\alpha} > 0, \rho \in (0,1), c \in (0,1)$; Set $\alpha \leftarrow \bar{\alpha}$;
repeat until $f(x_k + \alpha p_k) \le f(x_k) + c\alpha \nabla f_k^T p_k$
        $\alpha \leftarrow \rho\alpha$;
end (repeat)
Terminate with $\alpha_k = \alpha$.
```

```r
# <r code> ==================================================================== #
bfgs_back <- function (x, a, b) {                           # BFGS using backtracking
  h = solve(2 * t(a) %*% a)                                  # inverse of the hessian
  obj <- function(x, a, b) norm(a %*% x - b, type = "2")**2          # object value
  grad <- function(x, a, b) 2 * t(a) %*% (a %*% x - b)                   # gradient
  id = diag(1, nrow = nrow(x))                                     # identity matrix
  x_k = matrix(rep(0, length(x)))     # initializing at \mathbf{x}_{0} = \mathbf{0}
  i = 1                                                  # setting iterations counter
  alpha = 1                                              # initializing at \alpha = 1
            # setting \rho and the constant as 0.7 (to be used in the backtracking)
  rho = const = .7
  obj.value = obj(x_k, a, b)        # computing the object value for the first time
                              # empthy object to count how many times the algorithm
  count = numeric(1)          #         enters in the backtracking at each iteration
  p_k = -h %*% grad(x_k, a, b)                           # computing search direction
  norms = numeric(1)            # empthy object to keep the norms at each iteration
                              # initializing while loop with the stopping criterion
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
    while (obj(x_k[ , i] + alpha * p_k, a, b) >         # initializing backtracking
          obj.value[i] + const * alpha * t(grad(x_k[ , i], a, b)) %*% p_k) {
               # counting how many times we do the backtracking at each iteration
      count[i] = count[i] + 1
      alpha = rho * alpha                    # updating \alpha (doing the backtracking)
    }
    x_k = cbind(x_k, x_k[ , i] + alpha * p_k)   # computing the new \mathbf{x}_{k}
    i = i + 1                                            # updating iteration counter
           # computing the difference between \mathbf{x}_{k+1} and \mathbf{x}_{k}
    s = x_k[ , i] - x_k[ , i-1]
                                         # computing the difference between gradients
    y = grad(x_k[ , i], a, b) - grad(x_k[ , i-1], a, b)
    rhok = as.numeric(1/(t(y) %*% s))                               # computing \rho_{k}
    h = (id - rhok * s %*% t(y)) %*% h %*% (id - rhok * y %*% t(s)) +
      rhok * s %*% t(s)                                  # computing \mathbf{H}_{k+1}
    p_k = -h %*% grad(x_k[ , i], a, b)           # computing new search direction
    obj.value[i] = obj(x_k[ , i], a, b)    # computing and keeping new object value
    norms[i] = norm(x - x_k[ , i], type = "2")     # computing and keeping the norm
    count[i] = 0                 # setting the count to zero for the new iteration
  }
    # returning the \mathbf{x}_{k}'s, object values, norms and backtracking counts
  return(list(x = x_k[ , -1], obj.value = obj.value[-1], norms = norms[-1]
              , count = count))
} ; xks <- bfgs_back(x.ast, a, b)            # running the BFGS with backtracking
tail(xks$obj.value, 1)                                   # final objective value
tail(xks$norms, 1)                                                  # final norm
xks$count                 # number of times that we do backtracking at each iteration
# </r code> ==================================================================== #

[1] 1002.763
[1] 1.298771e-05
 [1] 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


# <r code> ===================================================================== #
par(mfrow = c(1, 2), mar = c(4, 4, 3, 2) + .1)             # graphical definitions
plot(xks$obj.value, log = "y", type = "b" # plotting the object values (log scale)
     , xlab = "Iteration", ylab = "Objective value (in log scale)", main = "(a)")
                                     # plotting norms with increasing iterations k
plot(xks$norms, type = "b", xlab = "Iteration", ylab = "L2 norm", main = "(b)")
# </r code> ==================================================================== #
```

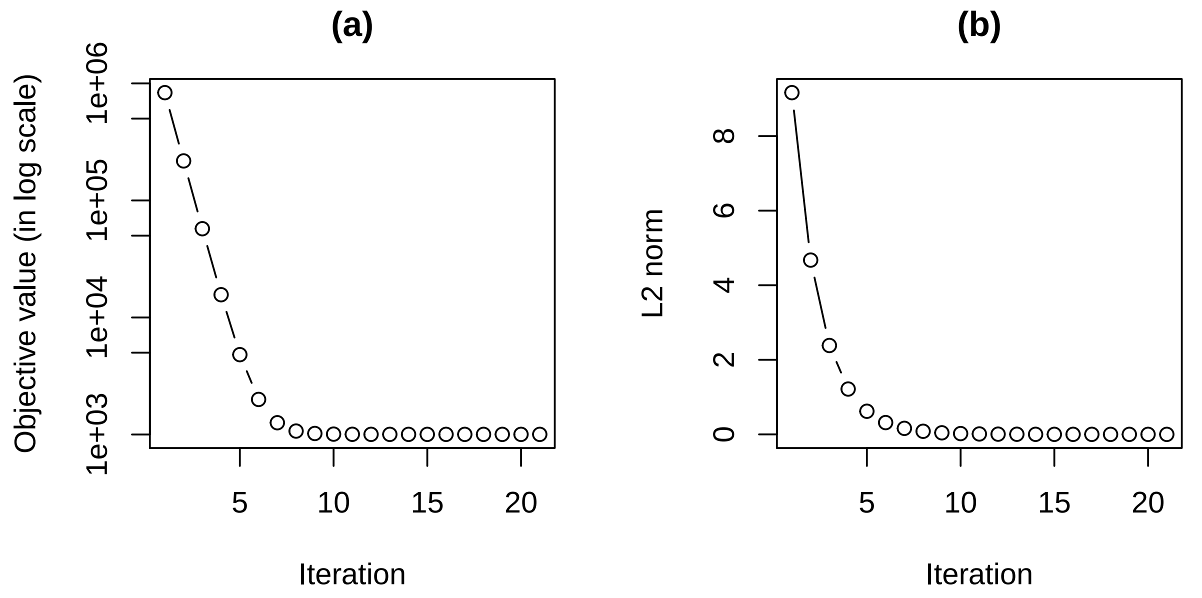


Figure 3: BFGS with backtracking. (a): The objective value (in log scale) with increasing k; (b): $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k.

Comparing with **(c)** - steepest descent with backtracking, we have the same final results (values). However, with BFGS with backtraking we see less iterations, 1/3 less, with the values decreasing more faster.

□

(e)

Implement Newton's method with exact line search. Use the same initialization and stopping criterion as in **(b)**. What do you notice?

Solution:

Newton's method iteration $k+1$ with exact line search:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\mathbf{p}_k^\top(\mathbf{A}^\top(\mathbf{A}\mathbf{x}_k - \mathbf{b}))}{\mathbf{p}_k^\top \mathbf{A}^\top \mathbf{A}\mathbf{p}_k}\mathbf{p}_k, \quad \text{with} \quad \mathbf{p}_k = -(\mathbf{A}^\top\mathbf{A})^{-1}(2\mathbf{A}^\top(\mathbf{A}\mathbf{x}_k - \mathbf{b})).$$

```
# <r code> ==================================================================== #
newton_els <- function(x, a, b) {          # newton's method with exact line search
  x_k = matrix(rep(0, length(x)))     # initializing at \mathbf{x}_{0} = \mathbf{0}
  i = 1                                                # setting iterations counter
  h_inv = solve(t(a) %*% a)                                # inverse of the hessian
                                   # initializing while loop with the stopping criterion
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
                                                       # computing search direction
    p_k =  - h_inv %*% (2 * t(a) %*% (a %*% x_k[ , i] - b))
    alpha = - (t(p_k) %*% (t(a) %*% (a %*% x_k[ , i] - b))) /
      (t(p_k) %*% t(a) %*% a %*% p_k)          # computing step length \alpha_{k}
                                               # computing the new \mathbf{x}_{k}
    x_k = cbind(x_k, x_k[ , i] + as.numeric(alpha) * p_k)
    i = i + 1                                          # updating iteration counter
  }
  return(xks = x_k[ , -1])                         # returning the \mathbf{x}_{k}'s
}
xks <- newton_els(x.ast, a, b) # running the newtons method with exact line search
norm(a %*% xks - b, type = "2")**2                       # computing objective value
norm(x.ast - xks, type = "2")                                      # computing norm
# </r code> =================================================================== #
```

```
[1] 1002.763
[1] 2.587757e-13
```

Newton's method for optimization converges in one step if the function is quadratic, as here. So here we have convergence in one interation. In **(b)** - steepest descent we have convergence after more than 20 iterations.

□

(f)

Implement the original and economic linear CG methods. Use the same initialization and stopping criterion as in **(b)**. Compare the performance of both methods w.r.t. the number of iterations needed to converge and the total time needed to converge. You can use the MATLAB commands *tic* and *toc* to measure the overall runtime.

Solution:

Original CG (Conjugate Gradient) method:

Algorithm 5.1 (CG–Preliminary Version).

```
Given $x_0$;
Set $r_0 \leftarrow Ax_0 - b$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$;
while $r_k \neq 0$
    $\alpha_k \leftarrow -\frac{r_k^T p_k}{p_k^T A p_k}$;             (5.14a)
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$;                          (5.14b)
    $r_{k+1} \leftarrow A x_{k+1} - b$;                               (5.14c)
    $\beta_{k+1} \leftarrow \frac{r_{k+1}^T A p_k}{p_k^T A p_k}$;     (5.14d)
    $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$;                  (5.14e)
    $k \leftarrow k + 1$;                                             (5.14f)
end (while)
```

```r
# <r code> =================================================================== #
cg_orig <- function (x, a, b) {          # original cg - conjugate gradient method
  x_k = matrix(rep(0, length(x)))    # initializing at \mathbf{x}_{0} = \mathbf{0}
  r_k = 2 * t(a) %*% (a %*% x_k - b)     # gradient = residual of the linear system
  p_k = -r_k                                                # initial search direction
  i = 1                                                   # setting iterations counter
  obj.value = norm(a %*% x_k - b, type = "2")**2                        # object value
  norms = numeric(1)             # empthy object to keep the norms at each iteration
  t1 = Sys.time()                                                       # initial time
                              # initializing while loop with the stopping criterion
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
                                                  # step length using exact line search
    alpha = - (t(r_k) %*% p_k) / (2 * t(p_k) %*% t(a) %*% a %*% p_k)
                                                     # computing the new \mathbf{x}_{k}
    x_k = cbind(x_k, x_k[ , i] + as.numeric(alpha) * p_k)
    i = i + 1                                           # updating iteration counter
    r_k = 2 * t(a) %*% (a %*% x_k[ , i] - b)                            # new residual
                                                      # computing the constant \beta_{k}
    beta_k = (t(r_k) %*% t(a) %*% a %*% p_k) / (t(p_k) %*% t(a) %*% a %*% p_k)
    p_k = -r_k + as.numeric(beta_k) * p_k                         # new search direction
                                                 # computing and keeping new object value
    obj.value[i] = norm(a %*% x_k[ , i] - b, type = "2")**2
    norms[i] = norm(x - x_k[ , i], type = "2")     # computing and keeping the norm
  }
 # returning the \mathbf{x}_{k}'s, object values and total time needed to converge
  return(list(x = x_k[ , -1], obj.value = obj.value[-1], norms = norms[-1]
              , time = Sys.time() - t1))
}                              # running the original cg - conjugate gradient method
xks_cg.orig <- cg_orig(x.ast, a, b)
# </r code> ============================================================== #
```

```r
# <r code> =============================================================== #
par(mfrow = c(1, 2), mar = c(4, 4, 3, 2) + .1)             # graphical definitions
                                                 # plotting the object values (log scale)
plot(xks_cg.orig$obj.value, log = "y", type = "b"
     , xlab = "Iteration", ylab = "Objective value (in log scale)", main = "(a)")
                                             # plotting norms with increasing iterations k
plot(xks_cg.orig$norms
     , type = "b", xlab = "Iteration", ylab = "L2 norm", main = "(b)")
# </r code> ============================================================== #
```

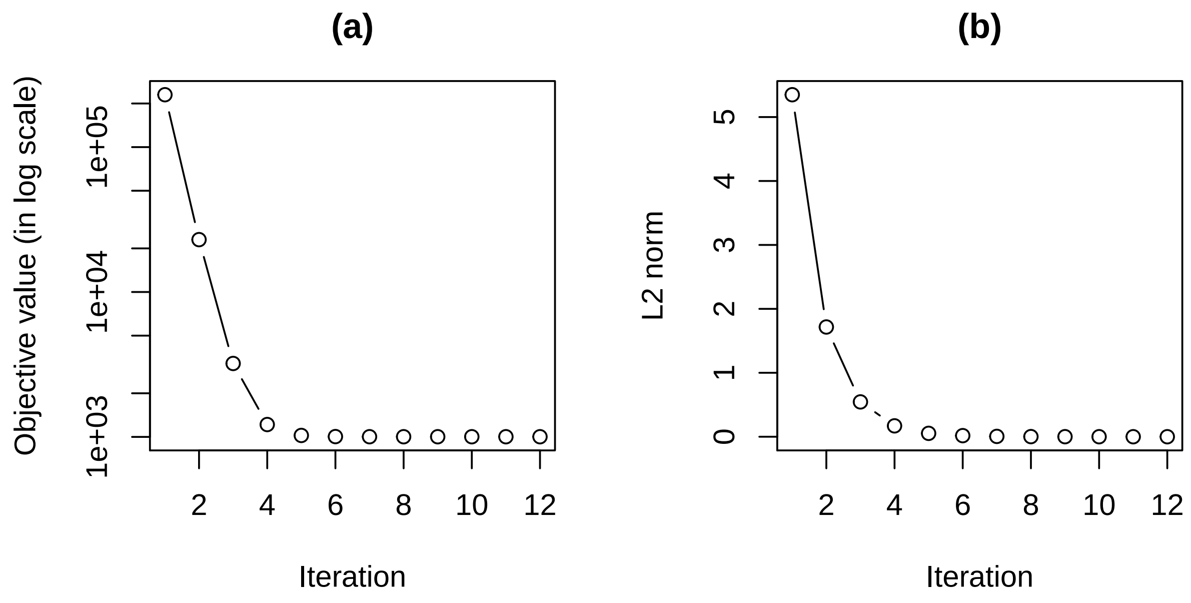


Figure 4: Original CG - conjugate gradient method. (a): The objective value (in log scale) with increasing k; (b): $\parallel \mathbf{x}^* - \mathbf{x}_k \parallel_2$ with increasing k.

Economic Linear CG (Conjugate Gradient) method:

Algorithm 5.2 (CG).

```
Given $x_0$;
Set $r_0 \leftarrow Ax_0 - b$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$;
**while** $r_k \neq 0$
```

$$\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}; \tag{5.24a}$$

$$x_{k+1} \leftarrow x_k + \alpha_k p_k; \tag{5.24b}$$

$$r_{k+1} \leftarrow r_k + \alpha_k A p_k; \tag{5.24c}$$

$$\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}; \tag{5.24d}$$

$$p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k; \tag{5.24e}$$

$$k \leftarrow k + 1; \tag{5.24f}$$

```
**end (while)**
```

```r
# <r code> ================================================================ #
cg_eco <- function (x, a, b) {    # economic linear cg - conjugate gradient method
  x_k = matrix(rep(0, length(x)))    # initializing at \mathbf{x}_{0} = \mathbf{0}
                                  # matrix to keep the residuals of two different iterations
  r_k = matrix(NA, ncol = 2, nrow(x))
                                                # gradient = residual of the linear system
  r_k[ , 1] = 2 * t(a) %*% (a %*% x_k - b)
  p_k = -r_k[ , 1]                                              # initial search direction
  i = 1                                                       # setting iterations counter
  obj.value = norm(a %*% x_k - b, type = "2")**2                          # object value
  norms = numeric(1)          # empthy object to keep the norms at each iteration
  t1 = Sys.time()                                                         # initial time
                                       # initializing while loop with the stopping criterion
  while (norm(x - x_k[ , i], type = "2") > 1e-6 * norm(x, type = "2")) {
                                                       # step length using exact line search
    heavy = t(a) %*% a %*% p_k
    alpha = (t(r_k[ , 1]) %*% r_k[ , 1]) / (2 * t(p_k) %*% heavy)
                                                          # computing the new \mathbf{x}_{k}
    x_k = cbind(x_k, x_k[ , i] + as.numeric(alpha) * p_k)
    i = i + 1                                                  # updating iteration counter
                                                                          # new residual
    r_k[ , 2] = r_k[ , 1] + as.numeric(alpha) * 2 * heavy
                                                           # computing the constant \beta_{k}
    beta_k = (t(r_k[ , 2]) %*% r_k[ , 2]) / (t(r_k[ , 1]) %*% r_k[ , 1])
    p_k = -r_k[ , 2] + as.numeric(beta_k) * p_k             # new search direction
                                          # computing and keeping new object value
    obj.value[i] = norm(a %*% x_k[ , i] - b, type = "2")**2
    norms[i] = norm(x - x_k[ , i], type = "2")     # computing and keeping the norm
    r_k[ , 1] = r_k[ , 2]           # setting the new residual as the old residual
  }
 # returning the \mathbf{x}_{k}'s, object values and total time needed to converge
  return(list(x = x_k[ , -1], obj.value = obj.value[-1], norms = norms[-1]
              , time = Sys.time() - t1))
}                     # running the economic linear cg - conjugate gradient method
xks_cg.eco <- cg_eco(x.ast, a, b)
# </r code> ==================================================================== #


# <r code> ===================================================================== #
par(mfrow = c(1, 2), mar = c(4, 4, 3, 2) + .1)             # graphical definitions
                                          # plotting the object values (log scale)
plot(xks_cg.eco$obj.value, log = "y", type = "b"
     , xlab = "Iteration", ylab = "Objective value (in log scale)", main = "(a)")
                                    # plotting norms with increasing iterations k
plot(xks_cg.eco$norms
     , type = "b", xlab = "Iteration", ylab = "L2 norm", main = "(b)")
# </r code> ==================================================================== #
```

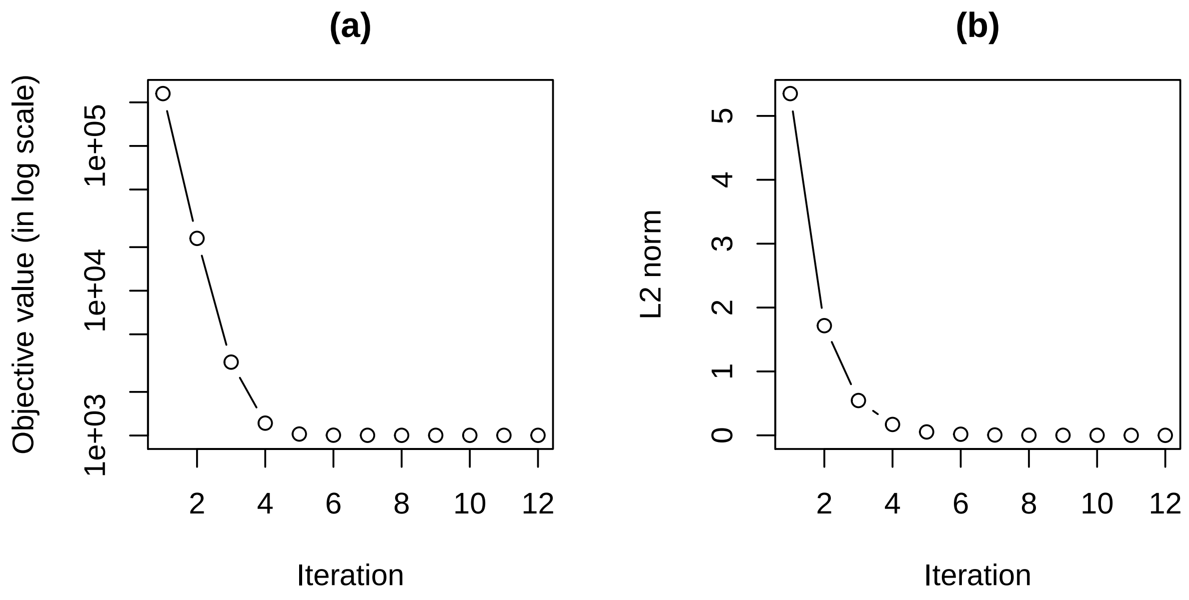


Figure 5: Economic Linear CG - conjugate gradient method. (a): The objective value (in log scale) with increasing k; (b): $\| \mathbf{x}^* - \mathbf{x}_k \|_2$ with increasing k.

Comparing:

```
# <r code> ==================================================================== #
                                              # original cg - conjugate gradient method
tail(xks_cg.orig$obj.value, 1)                                 # final objective value
tail(xks_cg.orig$norms, 1)                                                # final norm
xks_cg.orig$time                                        # total time needed to converge
# </r code> =================================================================== #
```

```
[1] 1002.763
[1] 1.576835e-05
Time difference of 4.048472 secs
```

```
# <r code> ==================================================================== #
                                       # economic linear cg - conjugate gradient method
tail(xks_cg.eco$obj.value, 1)                                  # final objective value
tail(xks_cg.eco$norms, 1)                                                 # final norm
xks_cg.eco$time                                         # total time needed to converge
# </r code> =================================================================== #
```

```
[1] 1002.763
[1] 1.576835e-05
Time difference of 1.492002 mins
```

Both methods reach the same values with the same number of iterations, 12. However, the original CG method need 4 seconds to converge, while the economic linear CG method need 1.49 minutes.

■