Line Search Methods

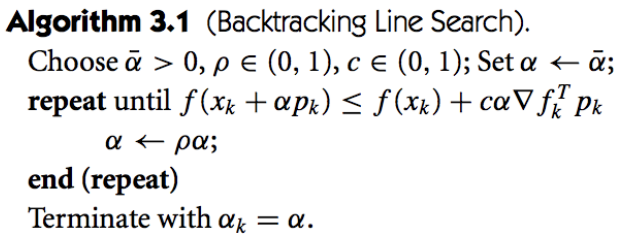
Henrique Ap. Laureano
ID 158811
KAUST

Spring Semester
2018

(a)

Solve Problem 3.1 on page 63 of Chapter 3 in textbook T1. MATLAB is preferred here. Show all your work, and submit your code. Choose your own and c. Plot the objective value with increasing iteration number. Use whichever stopping criterion you see fit.

Problem 3.1 Program the steepest descent and Newton algorithms using the backtracking line search, Algorithm 3.1. Use them to minimize the Rosenbrock function (2.22). Set the initial step length $\alpha_0 = 1$ and print the step length used by each method at each iteration. First try the initial point $x_0 = (1.2, 1.2)^\top$ and then the more difficult starting point $x_0 = (-1.2, 1)^\top$.



```
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 $\alpha_k = \alpha$.
```

$$\text{Rosenbrock function :} \quad f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \qquad (2.22)$$

Solution:

```
# <r code> =================================================================== #
exer3.1 <- function (x0, method) {      # initial points and specifying the method
  rosen <- function(x) {                                     # rosenbrock function
    fn = 100 * (x[2] - x[1]**2)**2 + (1 - x[1])**2         # computing the function
    grad = matrix(                          # computing the gradient, 2 x 1 matrix
      c( x[1] * (400 * x[1]**2 - 400 * x[2] + 2) - 2, 200 * (x[2] - x[1]**2) ) )
    hess = matrix(                           # computing the hessian, 2 x 2 matrix
```

```r
      c( 1200 * x[1]**2 - 400 * x[2] + 2, -400 * x[1], -400 * x[1], 200 ), 2, 2
      , byrow = TRUE)
    return(list(fn = fn, grad = grad, hess = hess))    # returning vectors/objects
  }
  direc <- function(obj, type) {                                # descent directions
    switch(type,
           steep = with(obj, - grad / norm(grad, type = "2")),  # steepest descent
           newton = with(obj, - solve(hess, grad))                        # newton
    )}
  x = matrix(x0)                 # converting the initial points to a 2 x 1 matrix
  ite = 100                                                 # number of iterations
  alpha = numeric(ite)                  # creating vector to keep the \alpha values
  alpha[1] = 1                                       # setting the first \alpha to 1
  rho = const = .5                                       # setting \rho and c to 0.5
  fn = numeric(ite)      # storing the rosenbrock function value at each iteration
  count = numeric(ite)      # storing the number of iterations in the backtracking
  rb = rosen(x)     # doing the rosenbrock function computations for the initial x
  p = direc(rb, method)        # computing the descent direction for the initial x
  for (i in 1:ite) {                                            # doing the loop
    while (rosen(x + alpha[i] * p)$fn >   # computing the backtracking line search
           rb$fn + const * alpha[i] * t(rb$grad) %*% p) {
      count[i] = count[i] + 1      # counting the number of times at each iteration
      alpha[i] = rho * alpha[i]                                 # updating the\ alpha
    }
    x = x + alpha[i] * p   # updating x with the (new) x and the descent direction
    rb = rosen(x)          # doing rosenbrock function computations for the new x
    fn[i] = rb$fn         # storing the rosenbrock function value in the iteration
    p = direc(rb, method)         # computing the descent direction for the new x
            # keeping the new \alpha to use again in the backtracking line search
    alpha[i + 1] = alpha[i]
            # returning the final x vector, the rosenbrock function value at each
            # iteration, the number of iterations at each backtracking iteration,
  }         #                             and the \alpha values at each iteration
  return(list(x = x, fn = fn, count = count, alpha = alpha[-ite+1]))
}
# </r code> ================================================================== #


# <r code> =================================================================== #
        # running the function in different scenarios (initial points and methods)
easy.steep  <- exer3.1(x0 = c(1.2, 1.2), method = "steep")
easy.newton <- exer3.1(x0 = c(1.2, 1.2), method = "newton")
hard.steep  <- exer3.1(x0 = c(-1.2, 1),  method = "steep")
hard.newton <- exer3.1(x0 = c(-1.2, 1),  method = "newton")
# </r code> ================================================================== #
# <r code> ==================================================================== #
                                                              # plotting the results
graph <- function(y, label, type, point) {

  title = switch(type
                 , alpha.steep = paste0(
                   "Steep length by Steepest descent\ninitial point "
                   , point)
                 , alpha.newton = paste0(
                   "Steep length by Newton algorithm\ninitial point "
                   , point)
                 , fn.steep = paste0(
                   "Objective value by Steepest descent\ninitial point "
                   , point)
                 , fn.newton = paste0(
                   "Objective value by Newton algorithm\ninitial point "
                   , point)
  )
  lattice::xyplot(y ~ 1:length(y), col = 1, type = "l", lwd = 3
                  , scales = list(x = list(at = c(1, seq(10 , length(y), 10))))
                  , xlab = "Iterations", ylab = label
                  , main = title)
}
print(graph(easy.steep$alpha,  expression(alpha), "alpha.steep",  "(1.2, 1.2)")
      , position = c(0, .75, .5,   1), more = TRUE)

print(graph(easy.newton$alpha, expression(alpha), "alpha.newton", "(1.2, 1.2)")
      , position = c(0,  .5, .5, .75), more = TRUE)

print(graph(hard.steep$alpha,  expression(alpha), "alpha.steep",  "(-1.2, 1)" )
      , position = c(0, .25, .5,  .5), more = TRUE)

print(graph(hard.newton$alpha, expression(alpha), "alpha.newton", "(-1.2, 1)" )
      , position = c(0,   0, .5, .25), more = TRUE)

print(graph(easy.steep$fn,  "f(x)", "fn.steep",  "(1.2, 1.2)")
      , position = c(.5, .75, 1,   1), more = TRUE)

print(graph(easy.newton$fn, "f(x)", "fn.newton", "(1.2, 1.2)")
      , position = c(.5,  .5, 1, .75), more = TRUE)

print(graph(hard.steep$fn,  "f(x)", "fn.steep",  "(-1.2, 1)" )
      , position = c(.5, .25, 1,  .5), more = TRUE)

print(graph(hard.newton$fn, "f(x)", "fn.newton", "(-1.2, 1)" )
      , position = c(.5,   0, 1, .25))
# </r code> =================================================================== #
```

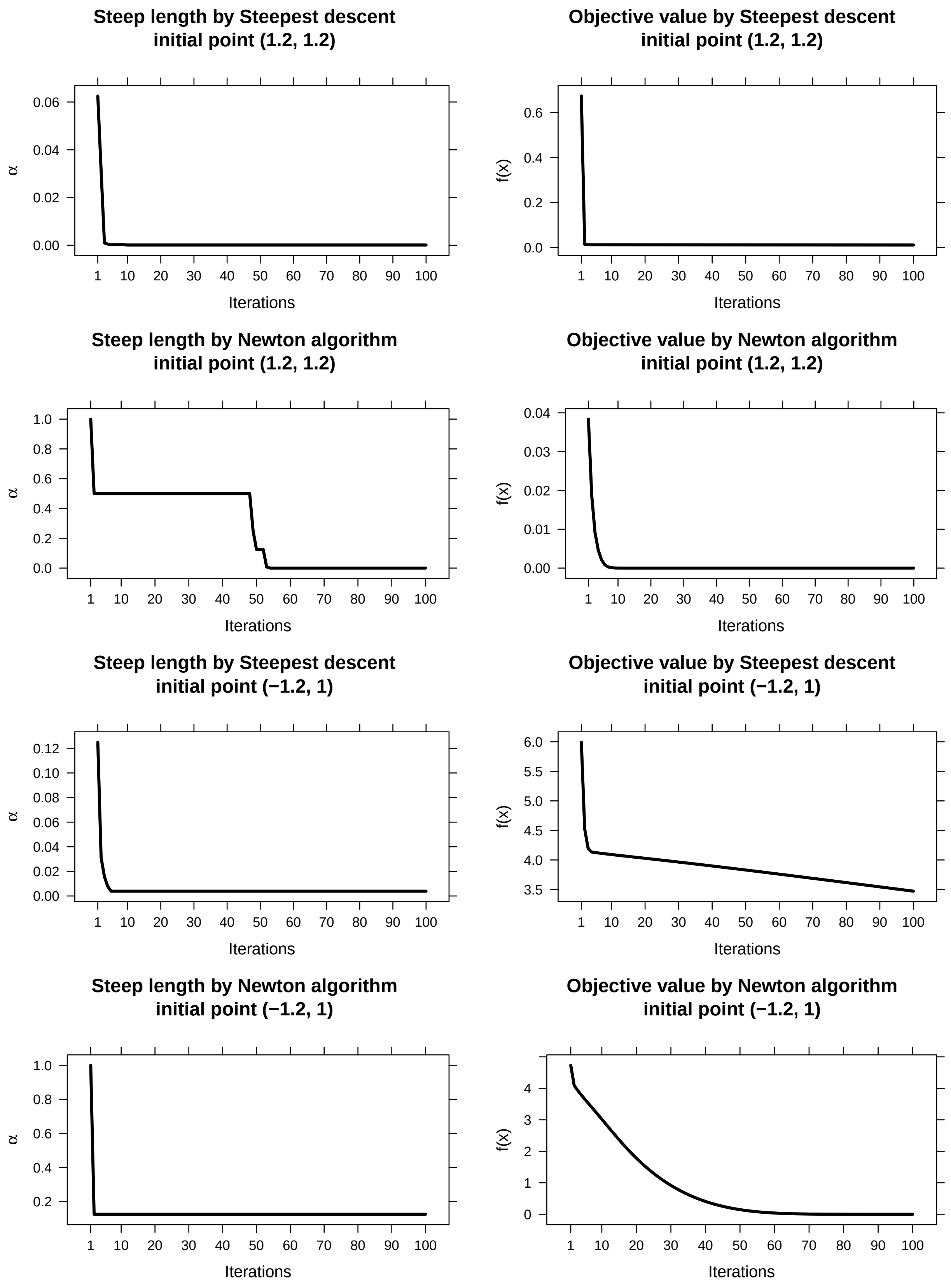


Figure 1: Steph length (at left) and objective value (at right) by each method at each (of 100) iteration(s), for an 'easy' and a more difficult starting point.

(b)

CVX is a MATLAB-based modeling system that supports disciplined convex programming. CVX is used to formulate and solve convex optimization problems. In this problem, we will employ this software to solve some data fitting problems. You can download the standard CVX package here. Provide your code for each of the subproblems below and print out the CVX solution you get.

CVXR Here I'm using `CVXR` (https://cran.r-project.org/web/packages/CVXR/index.html, https://cvxr.rbind.io/), a `R` version of CVX.

```
# <r code> =================================================================== #
library(CVXR) # laoding the library
# </r code> ================================================================== #
```

Let $\mathbf{A}$ and $\mathbf{b}$ be defined as follows:

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & 1 \\ -1 & 3 & 2 \\ 1 & -1 & 1 \end{bmatrix}; \quad \mathbf{b} = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

```
# <r code> =================================================================== #
A <- matrix(c( 3,  2,  1,                                  # creating the matrix A
             -1,  3,  2,
              1, -1, -1), 3, 3, byrow = TRUE)
b <- matrix(c(10, 5, -1), 3, 1)                            # creating the vector b
# </r code> ================================================================== #
```

i.

l_2 data fitting: Prove that problem **P1** is convex. Use CVX to solve **P1** given **A** and **b**. [Hint: Use **norm** to solve **P1**].

$$\mathbf{P1}: \quad \min_{\mathbf{x}} \| \mathbf{Ax} - \mathbf{b} \|_2$$

Solution:

Proof: $\| \cdot \|_2$ is a convex function and $\mathbf{Ax} - \mathbf{b}$ is affine. Therefore, **P1** is convex.

```
# <r code> =================================================================== #
x      <- Variable(3)                                                # creating x
obj    <- p_norm(A %*% x - b, 2)                                   # computing P1
prob   <- Problem(Minimize(obj))                # following the package procedure
result <- solve(prob)                                                   # solving
result[-c(8:9)]                                                 # showing results
# </r code> ================================================================== #
```

```
$status
[1] "optimal"

$value
[1] -1.548944e-11

$`2`
     [,1]
[1,]    2
[2,]    1
[3,]    2

$solver
[1] "ECOS"

$solve_time
[1] 5.2036e-05

$setup_time
[1] 4.8653e-05

$num_iters
[1] 5
```

x:

```
# <r code> ==================================================================== #
result[3]                                                                     # x
# </r code> =================================================================== #
```

```
$`2`
     [,1]
[1,]    2
[2,]    1
[3,]    2
```

ii.

Problem **P1** can be equivalently formulated as **P2**. Prove that **P2** is also convex and all its global solutions are the same as those of **P1**. Generate the equivalent quadratic form and use **quad_form** in CVX to solve it.

$$\mathbf{P2}: \quad \min_{\mathbf{y}} \| \mathbf{Ay} - \mathbf{b} \|_2^2$$

<u>Solution</u>:

$$\min_{\mathbf{y}} g(\mathbf{y}) = \| \mathbf{Ay} - \mathbf{b} \|_2^2 \quad \Rightarrow \quad \min_{\mathbf{y}} g(\mathbf{y}) = \mathbf{y}^\top \mathbf{A}^\top \mathbf{Ay} - 2(\mathbf{A}^\top \mathbf{b})^\top \mathbf{y} + \| \mathbf{b} \|_2^2$$

Proof The matrix $\mathbf{A}$ is full rank, so the Hessian $\nabla^2 g(\mathbf{y}) = 2\mathbf{A}^\top\mathbf{A}$ is positive definite. Therefore, $g(\mathbf{y})$ is convex. $f(\mathbf{x}) = \| \mathbf{Ax} - \mathbf{b} \|_2$ is convex, then $g(\mathbf{y}) = f(\mathbf{x})^2$ is also convex and the global optimum $\mathbf{x}^*$ satisfies $\nabla f(\mathbf{x}^*) = \nabla g(\mathbf{x}^*)$. The global solution is obtained when $\nabla g(\mathbf{y}) = 0$, $\mathbf{y} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$. Therefore, the solutions to **P1** and **P2** are identical.

```
# <r code> =================================================================== #
y      <- Variable(3)                                                # creating y
                                                                    # computing P2
obj    <- quad_form(y, t(A) %*% A) - 2 * t(t(A) %*% b) %*% y + p_norm(b, 2)**2
prob   <- Problem(Minimize(obj))                     # following the package procedure
result <- solve(prob)                                                   # solving
result[-c(8:9)]                                                 # showing results
# </r code> ================================================================== #
```

```
$status
[1] "optimal"

$value
[1] -7.58099e-07

$`2`
         [,1]
[1,] 2.000250
[2,] 1.000125
[3,] 2.000250

$solver
[1] "ECOS"

$solve_time
[1] 5.9781e-05

$setup_time
[1] 4.4631e-05

$num_iters
[1] 11
```

$\mathbf{y}$:

```
# <r code> =================================================================== #
result[3]                                                                   # y
# </r code> ================================================================== #
```

```
$`2`
         [,1]
```

```
[1,] 2.000250
[2,] 1.000125
[3,] 2.000250
```

iii.

l_1 *data fitting with a maximum value and* l_1 *regularizers*: Prove that **P3** is convex, if $\lambda \leq 0$ and $\mu \leq 0$. Solve the problem using CVX with $\lambda = \mu = 2$.

$$\mathbf{P3}: \quad \min_{\mathbf{z}} \parallel \mathbf{Az} - \mathbf{b} \parallel_1 + \mu \parallel \mathbf{z} \parallel_\infty + \lambda \parallel \mathbf{z} \parallel_1$$

Solution:

Proof $\| \cdot \|_1$ and $\| \cdot \|_\infty$ are both convex function and **P3** is a nonnegative combination of three convex functions. Therefore, **P3** is convex.

```
# <r code> ==================================================================== #
z      <- Variable(3)                                                # creating z
mu     <- lambda <- 2                                            # setting values
                                                                  # computing P3
obj    <- p_norm(A %*% z - b, 1) + mu * p_norm(z, Inf) + lambda * p_norm(z, 1)
prob   <- Problem(Minimize(obj))                      # following the package procedure
result <- solve(prob)                                                   # solving
result[-c(8:9)]                                                 # showing results
# </r code> =================================================================== #
```

```
$status
[1] "optimal"

$value
[1] 12.5

$`2`
     [,1]
[1,]  1.5
[2,]  1.5
[3,]  1.0

$solver
[1] "ECOS"

$solve_time
[1] 5.9584e-05

$setup_time
```

```
[1] 4.655e-05

$num_iters
[1] 7
```

z:

```
# <r code> ==================================================================== #
result[3]                                                                      # z
# </r code> =================================================================== #
```

```
$`2`
     [,1]
[1,]  1.5
[2,]  1.5
[3,]  1.0
```

■