MATH319 – Assignment 4

Pierre Visconti

February 21, 2024

Part (a):

Defining parameters

```
x0 <- c(1.2, 1.2)
c1 <- 0.4
rho <- 0.8
tol <- 0.000001
iter <- 500
```

Creating functions

```
# rosenbrock function
rosenbrock <- function(x1, x2) {
  out <- (x2-x1^2)^2 + (1-x1)^2
  return(out)
}
# compute the gradient
grad <- function(x1, x2) {
  c(-4*x1*(x2-x1^2)-2*(1-x1), 2*(x2-x1^2))
}
# compute the hess
hess <- function(x1, x2) {
  matrix(c(-4*x2+12*x1^2+2, -4*x1, -4*x1, 2), 2, 2)
}
```

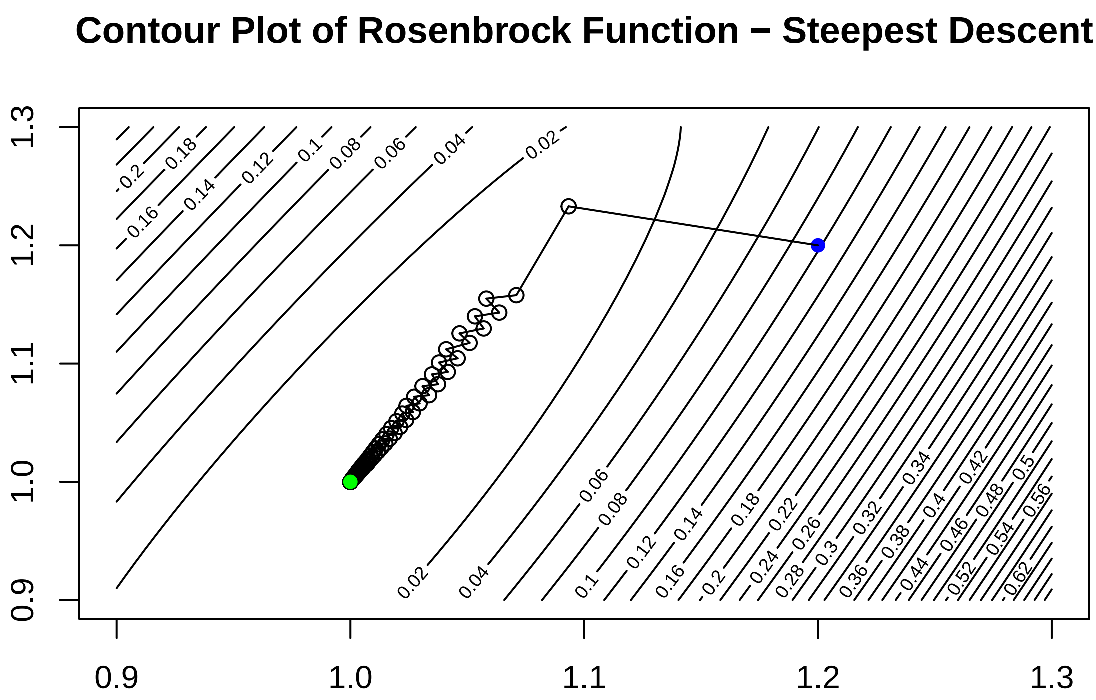
Steepest Descent:

```
# make a contour plot
x1 <- seq(0.9, 1.3, length.out=100)
x2 <- x1
z <- outer(x1, x2,FUN=rosenbrock)
plot_contour <- contour(x1, x2, z, nlevels=50,
                        main="Contour Plot of Rosenbrock Function - Steepest Descent")

# steepest descent function
steepest_descent <- function(func, x0) {
  # plot initial point x0
  points(x0[1], x0[2], col="blue", pch=16)

  # initialize variables
  xk <- x0
  norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
```

```r
  count <- 0

  # algorithm logic
  while(count < iter && norm_g > tol) {
    a <- 1
    # choose a descent direction
    pk <- -grad(xk[1], xk[2])
    # initialize step length variables
    x_ap <- xk + a*pk
    wolfe = func(x_ap[1], x_ap[2]) <
            (func(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
    # compute step length based on first Wolfe condition
    while (!wolfe) {
      # update step length
      a <- rho*a
      # update wolfe condition
      x_ap <- xk + a*pk
      wolfe = func(x_ap[1], x_ap[2]) <
              (func(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
    }
    # update iterate and norm
    xk_o <- xk
    xk <- xk + a*pk
    norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
    count <-  count + 1
    # plot iterate
    points(xk[1], xk[2])
    segments(xk_o[1], xk_o[2], xk[1], xk[2])
  }

  cat("Iterations: ", count, "\n")
  cat("Optimal value: ", xk)
}

steepest_descent(rosenbrock, x0)
```

```
## Iterations:  194
## Optimal value:  1.000001 1.000002
```

```r
# plot optimal solution
points(1,1, col="green", pch=16)
```

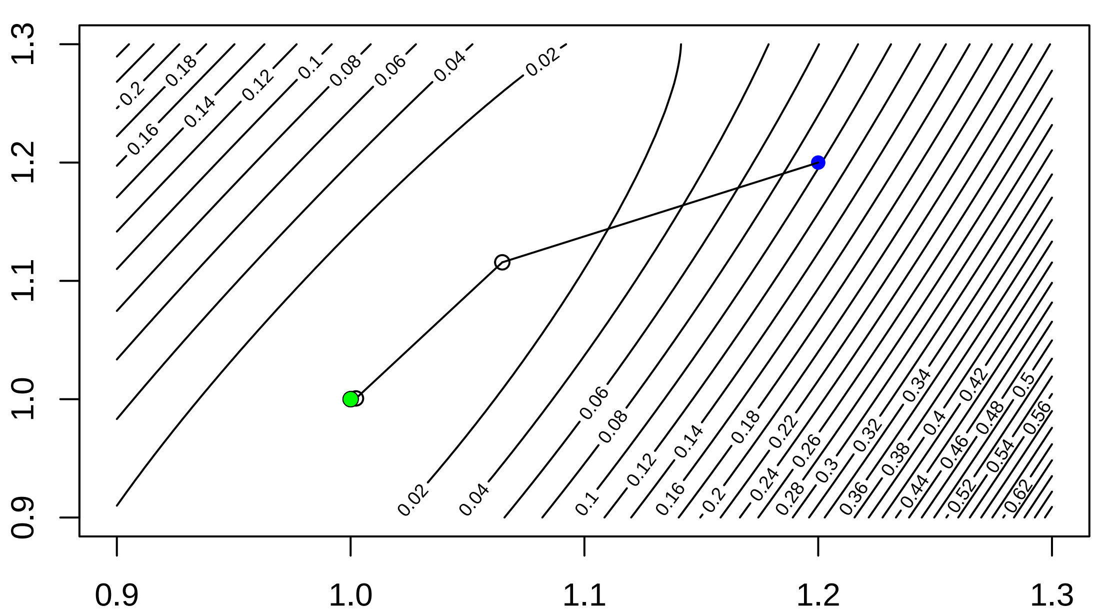


Newton Descent:

```
# make a contour plot
x1 <- seq(0.9, 1.3, length.out=100)
x2 <- x1
z <- outer(x1, x2,FUN=rosenbrock)
plot_contour <- contour(x1, x2, z, nlevels=50,
                        main="Contour Plot of Rosenbrock Function - Newton Descent")
# plot initial point x0
points(x0[1], x0[2], col="blue", pch=16)

# initialize variables
xk <- x0
norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
count <- 0

# algorithm logic
while(count < iter && norm_g > tol) {
  a <- 1
  # choose a descent direction, fix the hess if not PD
  if (all(eigen(hess(xk[1], xk[2]))$values > 0)) {
    pk <- -solve(hess(xk[1], xk[2])) %*% grad(xk[1], xk[2])
  } else {
    pk <- -solve(hess(xk[1], xk[2]) + tol*diag(2)) %*% grad(xk[1], xk[2])
  }
  # initialize step length variables
  x_ap <- xk + a*pk
  wolfe = rosenbrock(x_ap[1], x_ap[2]) <
          (rosenbrock(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
  # compute step length based on first Wolfe condition
  while (!wolfe) {
```

```r
    # update step length
    a <- rho*a
    # update wolfe condition
    x_ap <- xk + a*pk
    wolfe = rosenbrock(x_ap[1], x_ap[2]) <
            (rosenbrock(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
  }
  # update iterate and norm
  xk_o <- xk
  xk <- xk + a*pk
  norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
  count <-  count + 1
  # plot iterate
  points(xk[1], xk[2])
  segments(xk_o[1], xk_o[2], xk[1], xk[2])
}

# plot optimal solution
points(1,1, col="green", pch=16)
```

Contour Plot of Rosenbrock Function – Newton Descent




```r
cat("Iterations: ", count)
```

```
## Iterations:  4
```

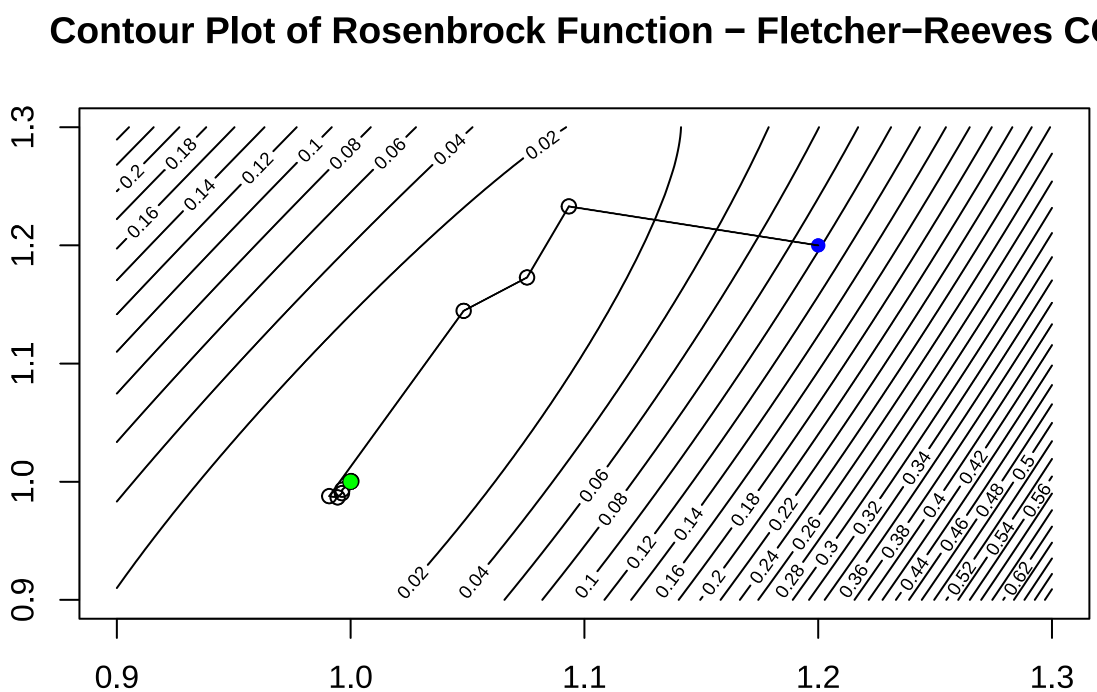
```r
cat("Optimal value: ", xk)
```

```
## Optimal value:  1 1
```

Fletcher-Reeves CG:

```r
# make a contour plot
x1 <- seq(0.9, 1.3, length.out=100)
x2 <- x1
```

```r
z <- outer(x1, x2,FUN=rosenbrock)
plot_contour <- contour(x1, x2, z, nlevels=50,
                        main="Contour Plot of Rosenbrock Function - Fletcher-Reeves CG")
# plot initial point x0
points(x0[1], x0[2], col="blue", pch=16)

# initialize variables
xk <- x0
grad_fk <- grad(xk[1], xk[2])
pk <- -grad_fk
norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
count <- 0

# algorithm logic
while(count < iter && norm_g > tol) {
  a <- 1
  # compute step length based on first Wolfe condition
  wolfe = rosenbrock(x_ap[1], x_ap[2]) <
          (rosenbrock(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
  while (!wolfe) {
    # update step length
    a <- rho*a
    # update wolfe condition
    x_ap <- xk + a*pk
    wolfe = rosenbrock(x_ap[1], x_ap[2]) <
            (rosenbrock(xk[1], xk[2]) + c1*a*t(grad(xk[1], xk[2]))%*%pk)
  }
  # update xk, norm, grad_fk, pk
  xk_o <- xk
  xk <- xk + a*pk
  norm_g <- norm(c(grad(xk[1], xk[2])[1], grad(xk[1], xk[2])[2]), type="2")
  grad_fk <- grad(xk[1], xk[2])
  pk <- -grad_fk + c(((t(grad_fk) %*% grad_fk)
                     %/% (t(grad(xk_o[1], xk_o[2])) %*% grad(xk_o[1], xk_o[2])))) * pk
  count <-  count + 1
  # plot iterate
  points(xk[1], xk[2])
  segments(xk_o[1], xk_o[2], xk[1], xk[2])
}

# plot optimal solution
points(1,1, col="green", pch=16)
```




```r
cat("Iterations: ", count)
```

```
## Iterations:  36
```

```r
cat("Optimal value: ", xk)
```

```
## Optimal value:  1 1
```

Discussion:

Steepest Descent took 194 iterations and obtained an optimal solution of (1.000001, 1.000002). Newton Descent took 4 iterations and obtained an optimal solution of (1,1). Fletcher-Reeves Conjugate Gradient took 36 iterations and obtained an optimal solution of (1,1). The worst performer, based on number of iterations and the fact that the exact solution was not reached, was Steepest Descent. While Newton Descent only took 4 iterations and FR-CG took 36, Newton Descent requires calculating the inverse of a matrix with every iteration while FR-CG does not, thus the actual computation time per iteration for FR-CG is significantly lower, especially for very large matrices. Therefore FR-CG was the best performer.

Part (b):

```r
# redefine the function, gradient, and hessian

# function
func_b <- function(x1, x2) {
  out <- (x2 - (5.1*x1^2)/(4*pi^2) + (5*x1)/pi - 6)^2 + 10*(1 - 1/(8*pi))*cos(x1) + 10
  return(out)
}
# compute the gradient
grad <- function(x1, x2) {
  a <- 2*(x2 - (5.1*x1^2)/(4*pi^2) + (5*x1)/pi - 6)*((-10.2*x1)/(4*pi^2) + 5/pi) -
        10*(1 - 1/(8*pi))*sin(x1)
  b <- 2*(x2 - (5.1*x1^2)/(4*pi^2) + (5*x1)/pi - 6)
```

```r
  return(c(a, b))
}
# compute the hess
hess <- function(x1, x2) {
  a <- 2*((-10.2*x1)/(4*pi^2) + 5/pi)^2 +
       2*(x2 - (5.1*x1^2)/(4*pi^2) + (5*x1)/pi - 6)*(-10.2/(4*pi^2)) -
       10*(1 - 1/(8*pi))*cos(x1)
  b <- 2*((-10.2*x1)/(4*pi^2) + 5/pi)
  d <- 2
  matrix(c(a, b, b, d), 2, 2)
}


# make a contour plot
x1 <- seq(-5, 15, length.out=100)
x2 <- x1
z <- outer(x1, x2,FUN=func_b)
plot_contour <- contour(x1, x2, z, nlevels=50,
                        main="Contour Plot of Function B - Steepest Descent")

# perform steepest descent with various starting points
steepest_descent(func_b, c(0, 6.1))
```

```
## Iterations:  198
## Optimal value:  -3.141592 12.275
```

```r
steepest_descent(func_b, c(0, 5.9))
```

```
## Iterations:  50
## Optimal value:  3.141593 2.275
```

```r
steepest_descent(func_b, c(6.5, 13))
```

```
## Iterations:  49
## Optimal value:  9.424778 2.475
```

```r
# find the function value for each optimal solution found
func_b(-3.141592, 12.275)
```

```
## [1] 0.3978874
```

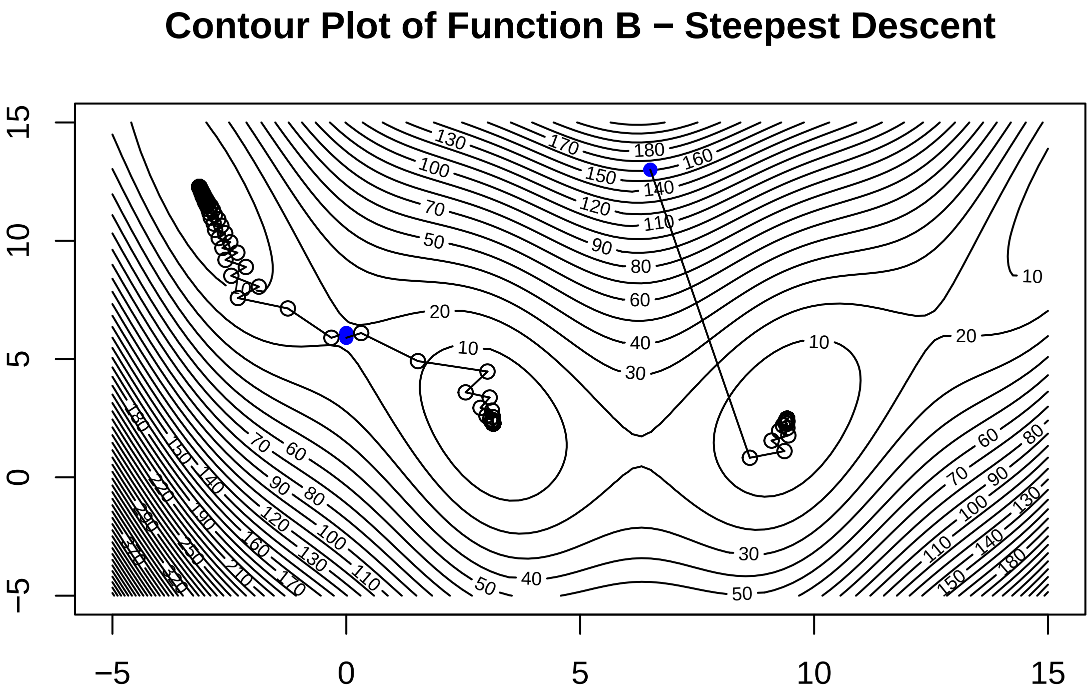
```r
func_b(3.141593, 2.275)
```

```
## [1] 0.3978874
```

```r
func_b(9.424778, 2.475)
```

```
## [1] 0.3978874
```

```r
points(-10,-10) # used to display plot as final output
```



Discussion:

The algorithm arrived to three different solutions, yet all three give the same optimal value when plugged back into the function, thus they are all optimal solutions.