From Scratch

Patrick Altmeyer

2020-10-13

Contents

Chapter 1

Introduction

Introduction

1.1 Some stuff

Chapter 2

Deterministic optimization

2.1 Line search

2.1.1 Methodology

The goal of Exercise 3.1 in **?** is to minimize the bivariate Rosenbrock function (Equation (**??**)) using *steepest descent* and *Newton's method*. The Rosenbrock function - also known as *Rosenbrock's banana function* - has a long, narrow, parabolic shaped flat valley and is often used for to test optimization algorithms for their performance (see here).

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{2.1}$$

We can implement Equation (**??**) in R as follows:

```
# Rosenbrock:
f = function(X) {
  100 * (X[2] - X[1]^2)^2 + (1 - X[1])^2
}
```

Figure **??** shows the output of the function over $x_1, x_2 \in [-1.5, 1.5]$ along with its minimum indicated as a red asterisk and the two starting points: (1) $X_0 = (1.2, 1.2)$ and (2) $X_0 = (-1.2, 1)$.

```
library(ggplot2)
# Plot
grid = data.table(expand.grid(x1=X_range,x2=X_range))
grid[,y:=f(c(x1,x2)),by=.(1:nrow(grid))]
X_min = grid[y==min(y),.(x1,x2)]
p = ggplot() +
```

```
  geom_contour_filled(data = grid, aes(x=x1,y=x2,z=y)) +
  geom_point(data = X_min, aes(x=x1,y=x2), colour="red", shape=8) +
  geom_point(data = X0, aes(x=x1,y=x2), colour="red") +
  geom_text(data = X0, aes(x=x1,y=x2,label=label), colour="red", nudge_x = 0.1, nudge_y
p
```

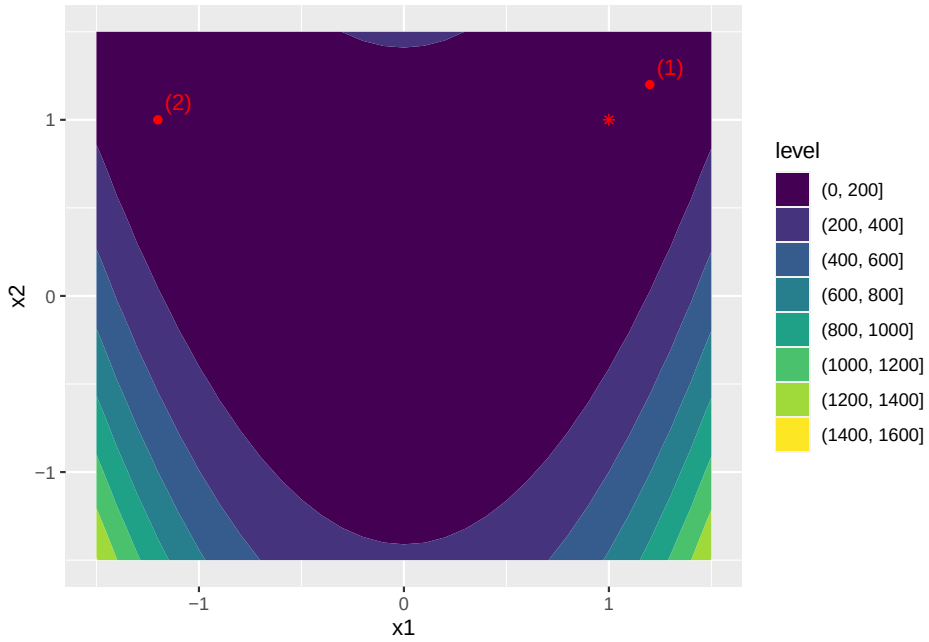

Figure 2.1: Output of the Rosenbrock function and minimizer in red.

The gradient and Hessian of f can be computed as

and

which in R can be encoded as follows:

```
# Gradient:
df = function(X) {
  df = rep(0, length(X))
  df[1] = -400 * X[1]^2 * (X[2] - X[1]^2) - 2 * (1 - X[1]) # partial with respect to x_
  df[2] = 200 * (X[2] - X[1]^2)
  return(df)
}

# Hessian:
ddf = function(X) {
  ddf = matrix(nrow = length(X), ncol=length(X))
  ddf[1,1] = 1200 * X[1]^2 - 400 * X[2] + 2 # partial with respect to x_1
```

```
  ddf[2,1] = ddf[1,2] = -400 * X[1]
  ddf[2,2] = 200
  return(ddf)
}
```

For both methods I will use the *Arminjo condition* with backtracking. The `gradient_desc` function (below) can implement both *steepest descent* and *Newton's method*. The code for the function can be inspected below (you can reveal it by simply clicking on the *Code* button on the right). There's also a small description of the different arguments.

```
#' Gradient descent
#'
#' @param f Function to be optimized.
#' @param df Gradient of function.
#' @param c Constant used in backtrack condition.
#' @param method Descent method.
#' @param X0 Initial guess.
#' @param step_size0 Initial step size.
#' @param ddf Hessian.
#' @param remember Boolean: should points visited and steps be stored?
#' @param tau Parameter governing the tolerance for convergence.
#' @param backtrack_cond Backtrack method.
#' @param max_iter Maximum number of iterations.
#'
#' @return
#' @export
#'
#' @examples
gradient_desc = function(
  f,df,X0,
  step_size0=1,
  ddf=NULL,
  method="newton",
  c=1e-4,
  remember=TRUE,
  tau=1e-5,
  backtrack_cond = "arminjo",
  max_iter=10000
) {
  # Initialization: ----
  X_latest = matrix(X0)
  if (remember) {
    X = matrix(X0,ncol=length(X0))
    steps = matrix(ncol=length(X0))
  }
```

```
  iter = 0
  # Set-up based on method:
  if (method=="steepest") {
    B = function(X) {
      diag(length(X)) # identity
    }
  } else if (method=="newton") {
    B = tryCatch(ddf, error=function(e) {
      stop("Hessian needs to be supplied for Newton's method.")
    }) # Hessian
  }
  # Backtrack condition
  if (backtrack_cond=="arminjo") {
    sufficient_decrease = function(alpha) {
      return(f(X_k + alpha * p_k) <= f(X_k) + c * alpha * t(df_k) %*% p_k) # Arminjo c
    }
  } else if (is.na(backtrack_cond)) {
    sufficient_decrease = function(alpha) {
      return(f(X_k + alpha * p_k) <= f(X_k)) # Standard condition
    }
  }
  # Run algorithm: ----
  while (any(abs(df(X_latest)-rep(0,length(X_latest)))>tau) & iter < max_iter) { # fir
    X_k = X_latest
    alpha = step_size0 # initialize step size
    df_k = matrix(df(X_latest))
    B_k = B(X_latest)
    p_k = - (solve(B_k) %*% df_k)
    # Backtracking:
    while (!sufficient_decrease(alpha)) {
      alpha = alpha/2
    }
    # Update:
    X_latest = X_latest + alpha * p_k
    iter = iter + 1
    if (remember) {
      X = rbind(X, t(X_latest))
      steps = rbind(steps, t(alpha * p_k))
    }
  }
  if (iter>=max_iter) warning("Reached maximum number of iterations without convergence
  # Tidy up: ----
  output = list(
    optimal = X_latest,
    visited = tryCatch(X, error=function(e) NULL),
```