# Metoda Newtona w optymalizacji

Autor: Jan Retkowski

### Importowanie bibliotek

```
library(ggplot2)
library(reshape2)
library(gridExtra)
library(ggthemes)
library(microbenchmark)
```

### Implementacja Metody Newtona przy znanym gradiencie i hessianie

Pierwszym krokiem jest zdefiniowanie funkcji `optimize.newton` która realizuje **Metodę Newtona** w optymalizacji oraz dwóch funkcji pomocniczych `prepare.plot2D` i `prepare.plot1D`, które zostaną wykorzystane do wizualizacji działania algorytmu.

```
optimize.newton <- function(x, func, gradient, hessian, iters, step_size, visual2D = F, visual1D = F) {
    h <- 1e-4

    if (visual2D) {
        g <- prepare.plot2D(x, func)
    } else if (visual1D) {
        g <- prepare.plot1D(x, func)
    }

    for (i in 1:iters) {

        d <- step_size * solve(hessian(x)) %*% gradient(x)

        if (visual2D) {

            g <- g + geom_segment(xend = x[1] - d[1], yend = x[2] - d[2], x = x[1],
                                  y = x[2], arrow = arrow(length = unit(0.2, "cm")),
                                  color = "red3", size = 0.3)
        } else if (visual1D) {
            g <- g + geom_segment(xend = x[1] - d[1], yend = drop(func(x[1] - d[1])),
                                  x = x[1], y = drop(func(x[1])),
                                  arrow = arrow(length = unit(0.2, "cm")),
                                  color = "red3", size = 0.3)
        }

        x <- x - d

        if (all(matrix(h, nrow = length(x), ncol=1) >= abs(x))) {
            break
```

```
        }

    }

    result <- list(min = NULL, max = NULL, value = func(x))

    if (hessian(x)[1, 1] > 0)
        result$min <- x
    else
        result$max <- x

    return( if (visual2D || visual1D) list(values = result, plot = g) else result)
}

prepare.plot2D <- function(x, func) {

    graph <- as.data.frame(expand.grid(seq(-abs(1.1 * x[1]), abs(1.1 * x[1]), length.out = 100),
                                       seq(-abs(1.1 * x[2]), abs(1.1 * x[2]), length.out = 100)))
    graph$z <- mapply(function(a, b) func(matrix(c(a, b), ncol = 1)), graph$Var1, graph$Var2)

    g <- ggplot(data = graph, aes(x = Var1, y = Var2, fill = z, z = z)) +
        geom_raster() +
        scale_fill_viridis_c() +
        geom_contour(color = "black", size = 0.2) +
        # geom_density2d() +
        geom_point(x = x[1], y = x[2], color = "red3", size = 1) +
        theme_wsj() +
        labs(x = "x", y = "y", fill = "z") +
        theme(legend.position = "right", legend.direction = "vertical", axis.title = element_text())

    return(g)
}

prepare.plot1D <- function(x, func) {

    graph <- data.frame(x = seq(-abs(1.1 * x[1]), abs(1.1 * x[1]), 1e-2))
    graph$y <- mapply(function(a) func(matrix(c(a), ncol = 1)), graph$x)

    g <- ggplot(data = graph, aes(x = x, y = y)) +
        geom_line() +
        geom_point(x = x[1], y = drop(func(x[1])), color = "red3", size = 1) +
        theme_wsj() +
        labs(x = "x", y = "y") +
        theme(legend.position = "right", legend.direction = "vertical", axis.title = element_text())

    return(g)
}
```

### Testowanie algorytmu

**Przypadek pierwszy** Następnie można przystąpić do testowania algorytmu. Pierwszą funkcją testową jest $f(x) = -x^\intercal x$. Jej gradient to $\nabla f(x) = -2x$, a hessian to $H_f = -2I_n$ gdzie $n = dim(x)$, a $I_n$ to

macierz diagonalna o wymiarach n na n. By sprawdzić działanie algorytmu, rezultat koncowy zostanie porównany do wyniku z oficjalnej implementacji Metody Quasi-Newtonowskiej wykorzystującej algorytm **Broyden–Fletcher–Goldfarb–Shanno** w **R** .

```
x <- matrix(c(20, 20), ncol = 1)

func <- function(x) -t(x) %*% x
gradient <- function(x) -2 * x
hessian <- function(x) -2 * diag(length(x))

cat("Newton's Method:", optimize.newton(x, func, gradient, hessian, 30, 0.5)$max, "\n")
```

```
## Newton's Method: 7.629395e-05 7.629395e-05
```

```
cat("R's BFGS:", optim(par = x, func, method = "BFGS", control = list(fnscale = -1))$par)  # fnscale =
```

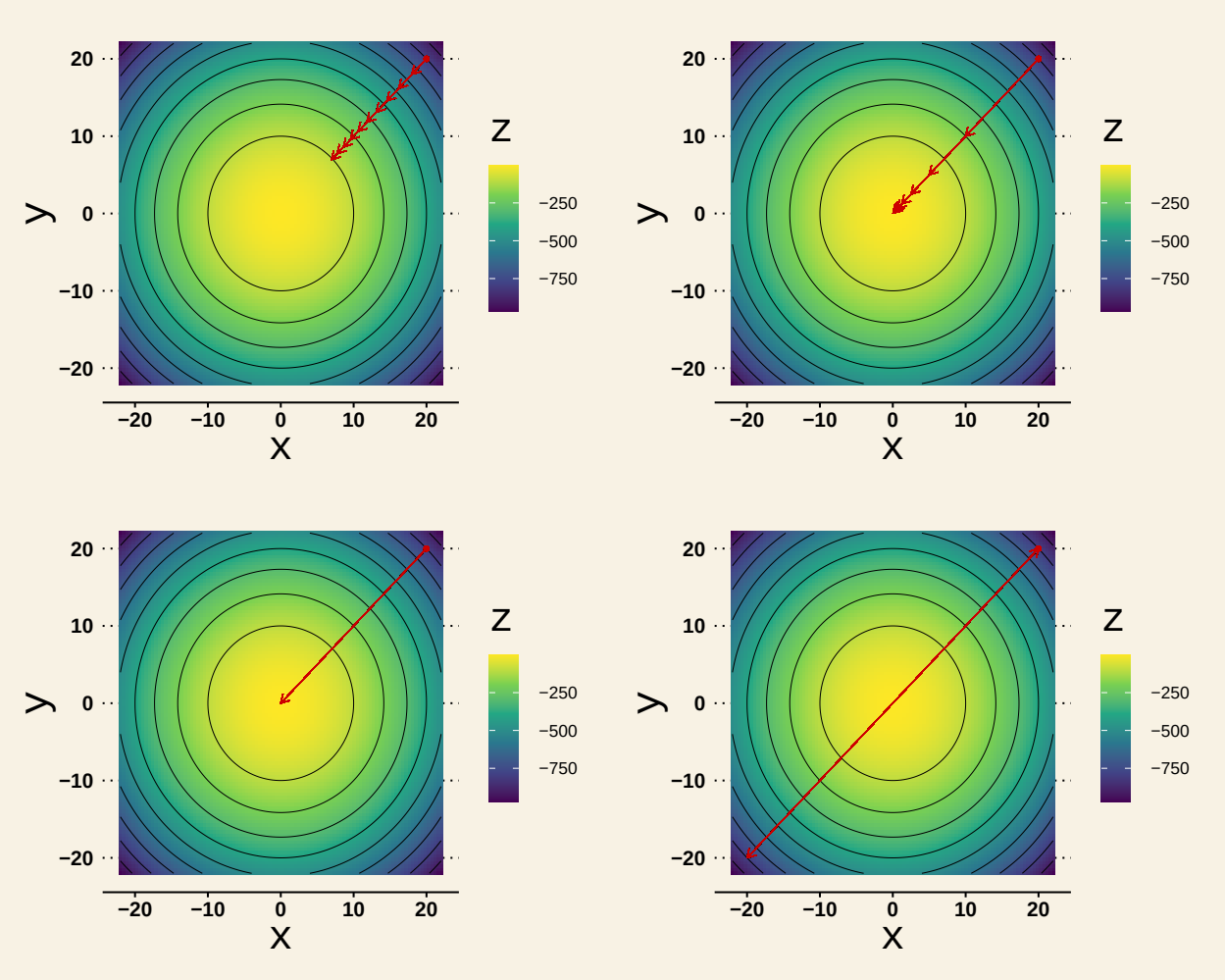
```
## R's BFGS: -2.750481e-16 -2.750481e-16
```

Jak widać obie funkcje zwróciły wartości bliskie wektorowi $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, w którym funkcja przyjmuje optimum globalne.

Teraz możliwe jest zwizualizowanie działania algorytmu. W tym celu maksymalna liczba króków `iters` zostanie zmniejszona, by wykres nie był był czytelniejszy oraz by budował się szybciej. Przy okazji zostanie zwizualizowane działanie algorytmu dla różnych wartości kroku `step_size`.

```
result1 <- optimize.newton(x, func, gradient, hessian, 10, 0.1, visual2D = T)
result2 <- optimize.newton(x, func, gradient, hessian, 10, 0.5, visual2D = T)
result3 <- optimize.newton(x, func, gradient, hessian, 10, 1, visual2D = T)
result4 <- optimize.newton(x, func, gradient, hessian, 10, 2, visual2D = T)

grid.arrange(result1$plot, result2$plot, result3$plot, result4$plot)
```




Rozmiar kroku wpływa znacząco na działanie elgorytmu. Zbyt mała wartość (w tym wypadku 0.1) sprawia, że algorytm bardzo wolno zbliża się do ekstremum. Z czasem dotarł by do celu, lecz w tym wypadku 10 iteracji nie wystarczyło. Przy warotści 0.5 algorytm dotarł do celu szybciej. Dla wartości 1 algorytm osiągnął ekstremum (punkt w któm gradient $\nabla f(x) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$) po zaletwie jednej iteracji. Natomiast wartość 2 okazała się stanowczo za duża, przez co algorytm przestrzelił wynik i oddalił się od ekstremum. Następny krok został skierowany w dobrym kierunku, lecz znów okazał się za duży. Wynika z tego, że metoda nigdy nie osiągnie pożądanego wyniku i będzie wokół niego oscylować.

Czas działania algorymu zostanie zbadany z wyłączoną opcją rysowania wykresów

```
microbenchmark(
    optimize.newton(x, func, gradient, hessian, 100, 0.1),
    optimize.newton(x, func, gradient, hessian, 100, 0.5),
    optimize.newton(x, func, gradient, hessian, 100, 1),
    optimize.newton(x, func, gradient, hessian, 100, 2),
    times = 100
)
```

```
## Unit: microseconds
##                                                  expr      min       lq
##  optimize.newton(x, func, gradient, hessian, 100, 0.1) 1802.889 1854.456
##  optimize.newton(x, func, gradient, hessian, 100, 0.5)  326.973  336.462
```

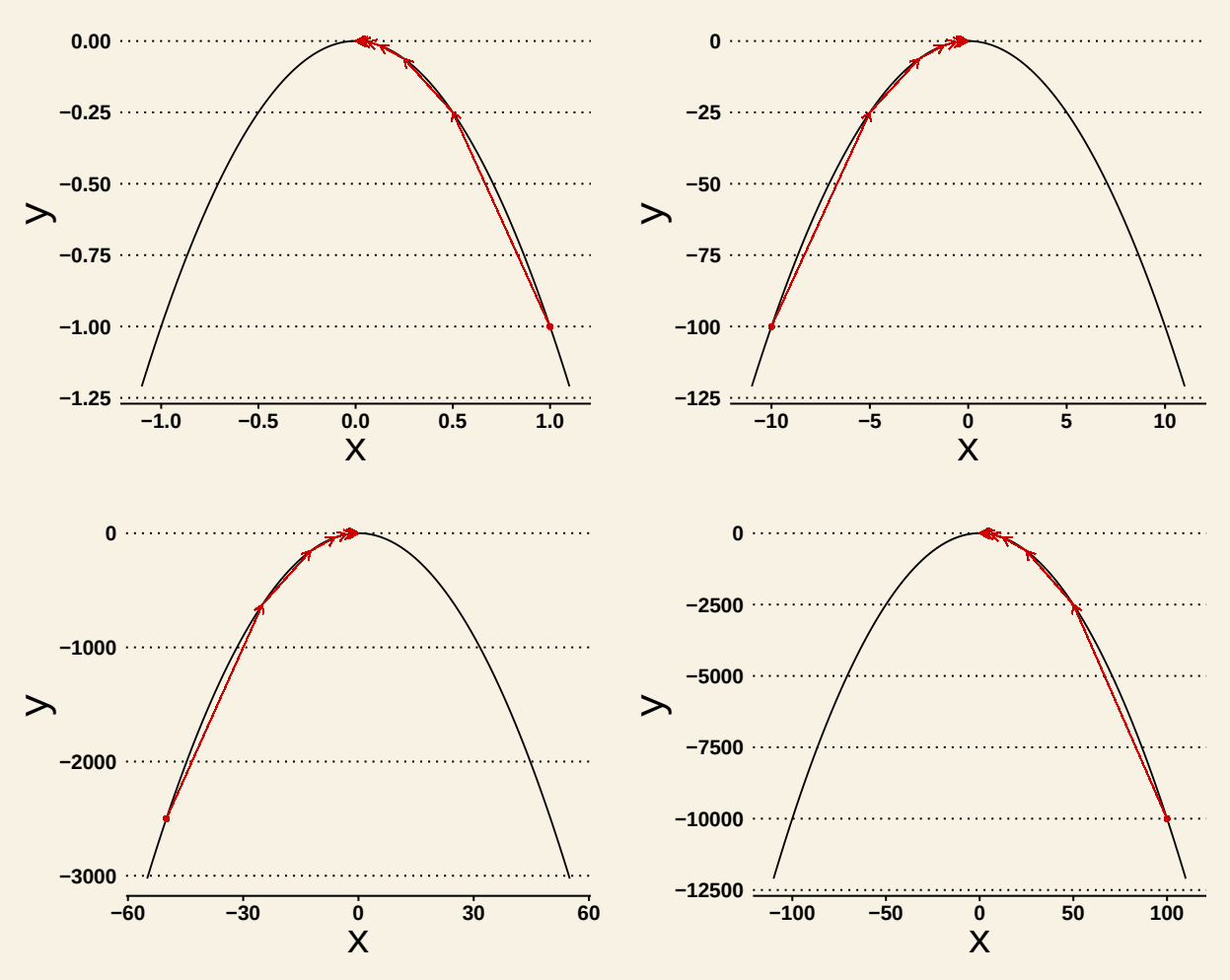
```
##    optimize.newton(x, func, gradient, hessian, 100, 1)   25.724   27.050
##    optimize.newton(x, func, gradient, hessian, 100, 2) 1801.611 1874.804
##        mean    median        uq      max neval
##  1981.46873 1914.0505 1968.4925 3472.166   100
##   412.80877  346.6035  360.9170 2026.680   100
##    32.40188   27.8860   30.6295  128.589   100
##  2039.70969 1918.2590 2031.1615 3614.932   100
```

Benchmark ukazuje, że wraz ze wzrostem watrości kroku (o ile nie przekracza on pewnej wartości), czas wykonania, a co za tym idzie liczba iteracji spada.

Dla tej funkcji zmiana punktu początkowego nie wpłynie znacząco na działanie metody. Oczywiście im dalej punkt będzie oddalony od ekstremum $\begin{pmatrix}0\\0\end{pmatrix}$ tym czas wykonania będzie dłuższy. Tym razem dla prezentacji wykorzystana zostanie jednowymiarowa przestrzeń poszuiwań.

```
result1 <- optimize.newton(c(1), func, gradient, hessian, 10, 0.5, visual1D = T)
result2 <- optimize.newton(c(-10), func, gradient, hessian, 10, 0.5, visual1D = T)
result3 <- optimize.newton(c(-50), func, gradient, hessian, 10, 0.5, visual1D = T)
result4 <- optimize.newton(c(100), func, gradient, hessian, 10, 0.5, visual1D = T)

grid.arrange(result1$plot, result2$plot, result3$plot, result4$plot)
```




Wykresy potwierdzają wcześniejsze przypuszczenia. Pomimo startowania z punktu $x_0 = 100$ metoda osiągnęła

ekstremum w podobnej liczbie iteracji co dla punktu startowego $x_0 = 1$. Wskazuje to iż kroki dla punktu $x_0 = 100$ musiały być początkowo znacznie większe.

Dla punktów startowych pomiary również zostaną wykonane bez rysowania wykresów.

```
microbenchmark(
    optimize.newton(c(1), func, gradient, hessian, 100, 0.5),
    optimize.newton(c(-10), func, gradient, hessian, 100, 0.5),
    optimize.newton(c(-50), func, gradient, hessian, 100, 0.5),
    optimize.newton(c(100), func, gradient, hessian, 100, 0.5),
    times = 100
)
```

```
## Unit: microseconds
##                                                      expr     min       lq
##    optimize.newton(c(1), func, gradient, hessian, 100, 0.5) 237.746 244.1505
##  optimize.newton(c(-10), func, gradient, hessian, 100, 0.5) 289.606 297.3010
##  optimize.newton(c(-50), func, gradient, hessian, 100, 0.5) 320.295 327.8960
##  optimize.newton(c(100), func, gradient, hessian, 100, 0.5) 336.919 350.4295
##      mean  median       uq      max neval
##  274.5802 256.388 273.7735  795.057   100
##  333.7550 310.879 325.2540 1540.857   100
##  366.2472 343.918 368.3740  716.741   100
##  370.9030 359.922 375.6475  575.461   100
```

Jak widać czasy są dla wszystkich punktów startowych są porównywalne.

**Przypadek drugi** Drugą funkcją poddaną testom będzie $f(x) = -x^\intercal x + 1.1\cos(x^\intercal x)$. Jej gradient to $\nabla f(x) = -2x - 2.2x\sin(x^\intercal x)$, a hessian to $H_f = -2I_n - 2.2I_n\sin(x^\intercal x) - 4.4xx^\intercal\cos(x^\intercal x)$.

```
x <- matrix(c(7), ncol = 1)

func <- function(x) -t(x) %*% x + 1.1 %*% cos(t(x) %*% x)
gradient <- function(x) -2 * x - 2.2 * x %*% sin(t(x) %*% x)
hessian <- function(x) {
   ( -2 * diag(length(x))
     - 2.2 * diag(length(x)) * drop(sin(t(x) %*% x))
     - 4.4 * x %*% t(x) * drop(cos(t(x) %*% x)))
}


cat("Newton's Method:", optimize.newton(x, func, gradient, hessian, 10, 0.1)$max, "\n")
```

```
## Newton's Method: 7.006134
```

```
cat("R's BFGS:", optim(par = x, func, method = "BFGS", control = list(fnscale = -1))$par)
```

```
## R's BFGS: 7.008865
```

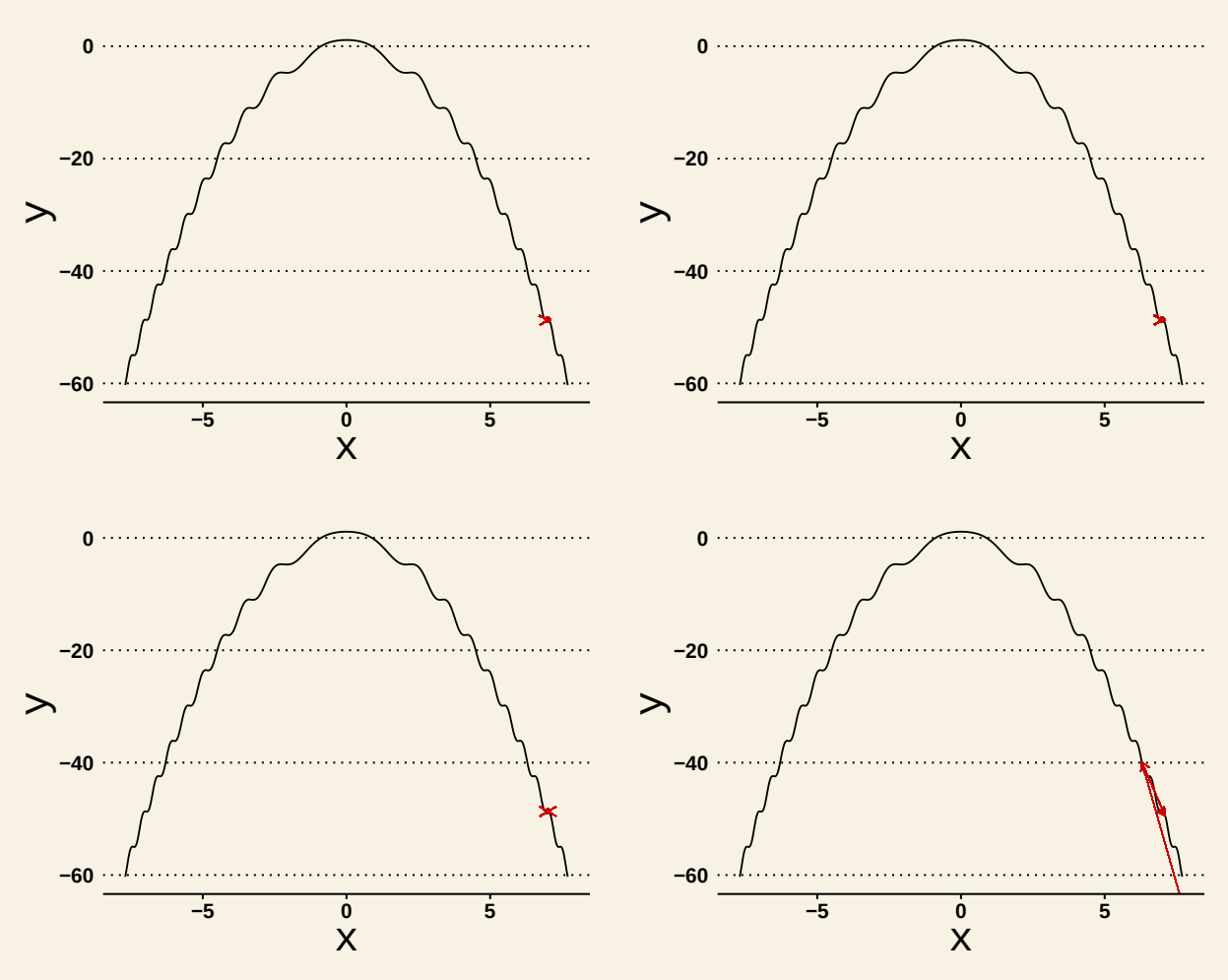
Dla tej funkcji Metoda Newtona praktycznie ni ruszyła się z miejsca startowego. Stało się tak zarówno dla testowanej implementacji, jak i dla funkcji `optim`. Przyczyną tego zjawiska jest, to iż fukcja $f(x) = -x^\intercal x + 1.1\cos(x^\intercal x)$ posiada nieskończoną ilośc ekstremów lokalnych, w których algorytm się zatrzymuje.

By lepiej zrozumieć to zjawisko dokonana zostanie wizualizacja, a zarazem sprawdzenie wpływu rozmiaru kroku na algorytm.

```
result1 <- optimize.newton(x, func, gradient, hessian, 10, 0.1, visual1D = T)
result2 <- optimize.newton(x, func, gradient, hessian, 10, 0.5, visual1D = T)
```

```
result3 <- optimize.newton(x, func, gradient, hessian, 10, 1.1, visual1D = T)
result4 <- optimize.newton(x, func, gradient, hessian, 10, 10, visual1D = T)

grid.arrange(result1$plot, result2$plot, result3$plot, result4$plot)
```




Dla mniejszych wartości kroku algorytm pozostaje w okolicach ekstremum lokalnego, które znajduje się bardzo blisko punktu początkowego $x = 7$. Natomiast dla większej wartości kroku (w tym wypadku `step_size = 10`) algorytm wydosaje się z okolic punktu początkowego, jednak zamiast zbliżać się do ekstremum globalnego, to fluktuje. Dzieje się tak, gdyż kierunek skoku zależy od tego, czy punkt znajdzie się w strefie chwilowego spadku, czy chwilowego wzrostu funkcji $f(x)$.

```
microbenchmark(
    optimize.newton(x, func, gradient, hessian, 100, 0.01),
    optimize.newton(x, func, gradient, hessian, 100, 0.5),
    optimize.newton(x, func, gradient, hessian, 100, 1.1),
    optimize.newton(x, func, gradient, hessian, 100, 10),
    times = 100
)
```

```
## Unit: milliseconds
##                                                  expr      min       lq
##  optimize.newton(x, func, gradient, hessian, 100, 0.01) 3.112781 3.256339
##   optimize.newton(x, func, gradient, hessian, 100, 0.5) 3.108728 3.252486
```

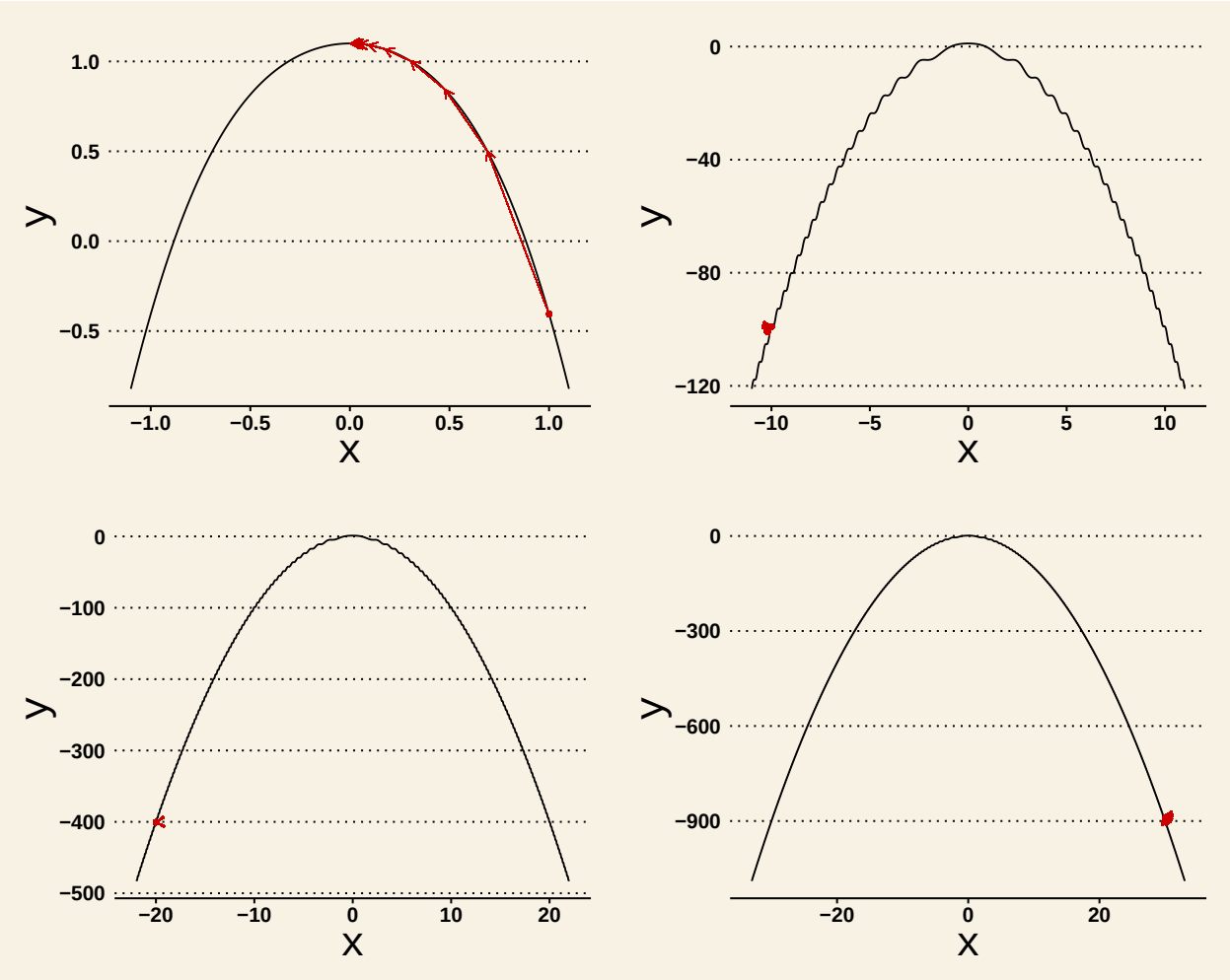
```
##   optimize.newton(x, func, gradient, hessian, 100, 1.1) 3.102510 3.233378
##    optimize.newton(x, func, gradient, hessian, 100, 10) 3.114224 3.265956
##      mean   median       uq      max neval
##  3.424008 3.346657 3.488605 4.788806   100
##  3.448459 3.335178 3.487564 4.954026   100
##  3.421473 3.297079 3.437383 4.685352   100
##  3.570277 3.368788 3.753235 5.137402   100
```

Czas wykonania nie zmienia się, gdyż algorytm we wszystkich przypadkach osiąga maksymalną dozwoloną liczbę kroków.

Nstępnie zwizualizowany zostanie wpływ punktu początkowego $x_0$ na działanie algorytmu dla tej funkcji.

```
result1 <- optimize.newton(c(1), func, gradient, hessian, 10, 0.5, visual1D = T)
result2 <- optimize.newton(c(-10), func, gradient, hessian, 10, 0.5, visual1D = T)
result3 <- optimize.newton(c(-20), func, gradient, hessian, 10, 0.5, visual1D = T)
result4 <- optimize.newton(c(30), func, gradient, hessian, 10, 0.5, visual1D = T)

grid.arrange(result1$plot, result2$plot, result3$plot, result4$plot)
```




Dla większości punktów startowych algorytm nie zdołał znaleźć ekstremum globalnego. Jedynym wyjątkiem jest punkt $x_0 = 1$, który znajduje się w bezpośrednim jego sąsiedztwie. W tym wypadku ekstremum globalne jest również, lokalnym co skutkuje znaleźeniem go przez metodę.

```r
microbenchmark(
    optimize.newton(c(1), func, gradient, hessian, 10, 0.5),
    optimize.newton(c(-10), func, gradient, hessian, 10, 0.5),
    optimize.newton(c(-20), func, gradient, hessian, 10, 0.5),
     optimize.newton(c(30), func, gradient, hessian, 10, 0.5),
    times = 100
)
```

```
## Unit: microseconds
##                                                    expr     min       lq
##    optimize.newton(c(1), func, gradient, hessian, 10, 0.5) 319.934 338.1705
##  optimize.newton(c(-10), func, gradient, hessian, 10, 0.5) 319.035 342.0835
##  optimize.newton(c(-20), func, gradient, hessian, 10, 0.5) 321.284 338.9395
##   optimize.newton(c(30), func, gradient, hessian, 10, 0.5) 321.776 341.4450
##      mean   median       uq      max neval
##  361.8389 345.6360 358.7560 1090.803   100
##  362.4045 349.0000 357.9320  735.318   100
##  350.8184 348.0725 357.6235  471.797   100
##  381.6114 350.6995 369.6310 1585.890   100
```

Punkt startowy również nie ma znaczącego wpływu na czas wykonania.

**Wnioski** Na posdstawie obserwacji można stwierdzić, iż **Metoda Newtona** służy do lokalizowania ekstremów lokalnych. Co za tym idzie nie sprawuje się dobrze w gdy celem jest globalna optymalizacja funkcji z dużą ilością ekstremów lokalnych.

---

### Implementacja algorytmu Broyden–Fletcher–Goldfarb–Shanno

Pochodną **Metody Newtona** (tzw. Metodą Quasi-Newtonowską) jest algorytm **Broyden–Fletcher–Goldfarb–Shanno**, czyli w skrócie **BFGS**. Główna różnica między tymi metodami polaga na sposobie wyznaczania gradientu i hessianu. Podczas gdy w **Metodzie newtona** gradient i hessian wyznaczane są formalnie przez "człowieka", algorytm **BFGS** szacuje je dynamicznie na podstawie danych stosując metody numeryczne.

```r
optimize.BFGS <- function(x, cost_func, iters, step_size, maximum = F) {

    func <- if(maximum) function(a) -cost_func(a) else cost_func

    h <- 1e-4

    line_equation <- function(from, to, p) from + p * (to - from)
    step_space <- seq(h, step_size, length.out = 10000)

    epsilon <- .Machine$double.eps
    func.vect <- Vectorize(func)
    hessian <- diag(length(x))
    gradient <- function(x) as.matrix((func.vect(x + h) - func.vect(x - h)) / (2 * h), ncol = 1)

    prev_gradient <- gradient(x)

    search_space <- lapply(step_space,
                           function (p) line_equation(x, x - step_size * prev_gradient, p))
```

```
    step_size_k <- step_space[which.min(mapply(func, search_space))]

    s <- -step_size_k * prev_gradient

    x <- x + s

    for (i in 1:iters) {

        prev_gradient <- gradient(x)

        d <- solve(hessian) %*% prev_gradient

        search_space <- lapply(step_space, function (p) line_equation(x, x - step_size * d, p))
        step_size_k <- step_space[which.min(mapply(func, search_space))]

        s <- -step_size_k * d

        x <- x + s

        y <- gradient(x) - prev_gradient

        hessian <- (hessian + (y %*% t(y)) / drop(t(y) %*% s)
                    - (hessian %*% s %*% t(s) %*% t(hessian)) / drop(t(s) %*% hessian %*% s))


        if (all(matrix(h, nrow = length(x), ncol=1) >= x)) {
            break
        }

    }

    result <- list(min = NULL, max = NULL, value = func(x))

    if (maximum)
        result$max <- x
    else
        result$min <- x

    return(result)
}
```

**Testowanie algorytmu** By pokazać, że mimo stosowania szacowań, metoda **BFGS** działa równie dobrze jak **Metoda Newtona** obie z nich zostaną wywołane dla identycznych danych.

```
x <- matrix(sample(-100:100, 4), ncol = 1)

func <- function(x) -t(x) %*% x
gradient <- function(x) -2 * x
hessian <- function(x) -2 * diag(length(x))

cat("Newton's Method:", optimize.newton(x, func, gradient, hessian, 50, 0.3)$max, "\n")
```

```
## Newton's Method: -2.364814e-05 6.184897e-05 -1.364316e-05 -7.276349e-05
```

```r
cat("BFGS: ", optimize.BFGS(x, func, 2, 1, maximum = T)$max, "\n")
```

```
## BFGS:  -4.76456e-22 -2.646978e-23 1.952146e-21 7.610062e-22
```

```r
cat("R's BFGS:", optim(par = x, func, method = "BFGS", control = list(fnscale = -1))$par)
```

```
## R's BFGS: -7.658984e-10 -1.827242e-09 -7.304511e-10 -1.167279e-09
```

Jak widać, wszystkie trzy funkcje zwróciły poprawyne wyniki, gdyż maksimum globalnym jest wektor $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.