MTH210: Lab 9 Solutions

Newton-Raphson and Gradient Ascent

1. **Go through the code in `classDemonstration.R` thoroughly. This contains the code for the location Cauchy example and for the logistic regression example.**

   Solutions for this are not necessary.

2. **Using both Newton-Raphson and gradient ascent algorithm, maximize objective function**

$$f(x) = \cos(x) \quad x \in [-\pi, 3\pi].$$

Our task is to find:

$$x^* = \arg \max_{x \in [-\pi, 3\pi]} \cos(x)$$

In this simple problem, we actually already know that the maxima occurs at two points: $x = 0, 2\pi$, both. Thus our algorithms are expected to converge to either of those two points. In order to implement Newton-Raphson, we need the gradient and the second derivative of the objective function. Let's find that first.

$$f'(x) = -\sin(x) \qquad f''(x) = -\cos(x)$$

Note that since $\cos(x)$ takes both positive and negative values from $[-\pi, 3\pi]$, the objective function is not concave. This means that Newton-Raphson can converge to a local minima as well!

Let us write functions to calculate the gradient and second derivative.

```r
f.grad <- function(x)  -sin(x)
f.hessian <- function(x) -cos(x)
```

Since the function is simple, the gradients and derivatives are simple to code, as seen above. Now let us first implement Newton-Raphson. Since Newton-Raphson is not guaranteed to converge, we have to be smart about choosing the starting value.

```r
tol <- 1e-10
compare <- 100
iter <- 1

xk <- c()  # will store sequence here
xk[1] <- .5 # starting value
while(compare > tol)
{
  iter <- iter + 1  # tracking iterations
  gradient <- f.grad(xk[iter - 1])
  hessian <- f.hessian(xk[iter - 1])
  xk[iter] <- xk[iter - 1] - gradient/hessian

  compare <- abs(gradient)
}
iter
```

```
[1] 5
```

```r
xk[iter] # N-R last iterate.
```

```
[1] 0
```

Starting from $x_0 = .5$, the above NR converges to 0. If we change the starting value, to say $\pi/2$, then since $\pi/2$ is an inflection point, the second derivative is 0 and the NR algorithm becomes unstable! (try it). Also change the starting value to a little more than $\pi/2$, say, $\pi/2 + .1$, and you will notice that the algorithm converges to a minima.

Now let's implement the gradient ascent algorithm. Here we need to choose a value of the learning rate, t. After playing around with the value, I find $t = 1$ is reasonable. For GA, it becomes essential to also store the gradients in order to understand whether the value of t chosen is ok or not. So we store the gradients as well. Further, since GA can take many loops to converge, we also put a check to stop if the number of loops exceed a certain value.

```r
tol <- 1e-10
compare <- 100
iter <- 1

xk <- c()  # will store sequence here
gradient <- c()  # will store gradient sequence here
xk[1] <- .5 # starting value
gradient[1] <- f.grad(xk[1])
t <- 1
```

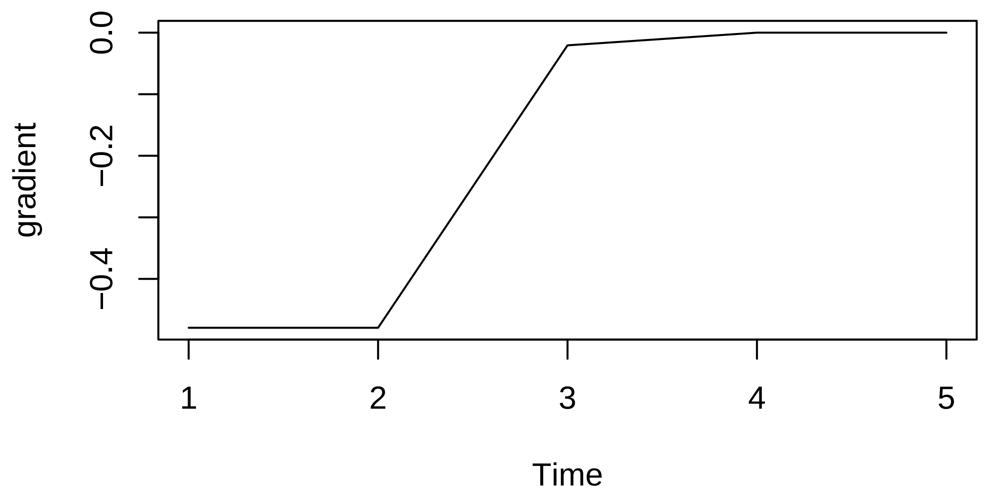
```r
while(compare > tol && iter < 100)
{
  iter <- iter + 1  # tracking iterations
  gradient[iter] <- f.grad(xk[iter - 1])
  xk[iter] <- xk[iter - 1] +  t*gradient[iter]

  compare <- abs(gradient[iter])
}
iter
```

```
[1] 5
```

```r
xk[iter] # GA last iterate.
```

```
[1] 0
```

```r
plot.ts(gradient) ##plot of gradients wrt to time
```



You can see that here also the algorithm converges to 0, and the gradients are very well behaved. Now change the value of t to be 10 and see what happens!

3. **Load the `titanic.csv` dataset that is available on my website:**

```r
titanic <- read.csv("https://dvats.github.io/assets/titanic.csv")
head(titanic)
```

```
  Survived X.Intercept. Sexmale Age SibSp Parch    Fare
1        0            1       1  22     1     0  7.2500
2        1            1       0  38     1     0 71.2833
3        1            1       0  26     0     0  7.9250
4        1            1       0  35     1     0 53.1000
```

```
5          0              1         1   35       0      0   8.0500
6          0              1         1   54       0      0 51.8625
```

This dataset has:

- **`Survived`: whether survived or alive**
- **`X.Intercept`: a column of 1s for the intercept.**
- **`Sexmale`: indicating whether the person is male (1) or female (0)**
- **`Age`: age**
- **`SibSp`: Number of Siblings/Spouses Aboard**
- **`Parch`: Number of Parents/Children Aboard**
- **`Fare`: Cost of the ticket.**

Consider fitting a logistic regression model for the `Survived` variable (y_i):

$$\Pr(Y_i = 1) = \frac{e^{x_i^T\beta}}{1 + e^{x_i^T\beta}}$$

Implement the Newton-Raphson algorithm to estimate the maximum likelihood estimator of β.

To implement Newton-Raphson's algorithm, we require to calculate both the gradient vector and the Hessian for this problem. Recall that these have been calculated in class already. From our notes, we know that:

$$\nabla l(\beta) = \sum_{i=1}^{n} x_i \left[y_i - \frac{1}{1 + e^{-x_i^T\beta}} \right]$$

The way we code this efficiently is that we for i, we calculate $(1 + e^{-x_i^T\beta})^{-1}$ in one vector which we call `pi.vec`. Then we multiple matrix of covariates X with the appropriate thing, and do a colSums.

```r
f.gradient <- function(y, X, beta)
{
  # converting beta to compatible matrix form
  beta <- matrix(beta, ncol = 1)
  pi.vec <-  1 / (1 + exp(-X%*%beta))
  rtn <- colSums(X* as.numeric(y - pi.vec))
  return(rtn)
}
```

The Hessian matrix was obtained to be

$$\nabla^2 l(\beta) = -X^T W X$$

where W has diagonals $e^{x_i^T\beta}/(1+e^{x_i^T\beta})^2$. Since the above matrix is negative semi-definite, we also know that the objective function is concave.

```
f.hessian <- function(y, X, beta)
{
  beta <- matrix(beta, ncol = 1)
  W_i <-  exp(X%*%beta) / (1 + exp(X%*%beta))^2
  W <- diag(as.numeric(W_i))
  rtn <- - t(X) %*% W %*% X
}
```

Now that these functions are set, we can resume implement the estimation procedure. First let's prepare the dataset, so that `y` and `X` are separate:

```
y <- titanic$Survived
X <- as.matrix(titanic[, -1]) # everything but the first column is the X

# will need these later
p <- dim(X)[2]
n <- length(y)
```

Now, the data is set ready, and we are all set to implement the the Newton-Raphson procedure. In the code below, we do not store all the sequence of βs (you can if you want to), but I store the norm of the gradient vector to show you that think work well for this, since the gradient is nicely converging to zero.

```
tol <- 1e-10
compare <- 100
iter <- 1

# starting from the zero-vector
grad.vec <- c() # will store gradients here
beta.current <- rep(0, p)
beta.new <- beta.current
while(compare > tol)
{
  iter <- iter + 1  # tracking iterations

  gradient <- f.gradient(y, X, beta.current)
  hessian <- f.hessian(y, X, beta.current)
  beta.new <- beta.current - qr.solve(hessian) %*% gradient

  grad.vec[iter] <- norm(gradient, "2")
```

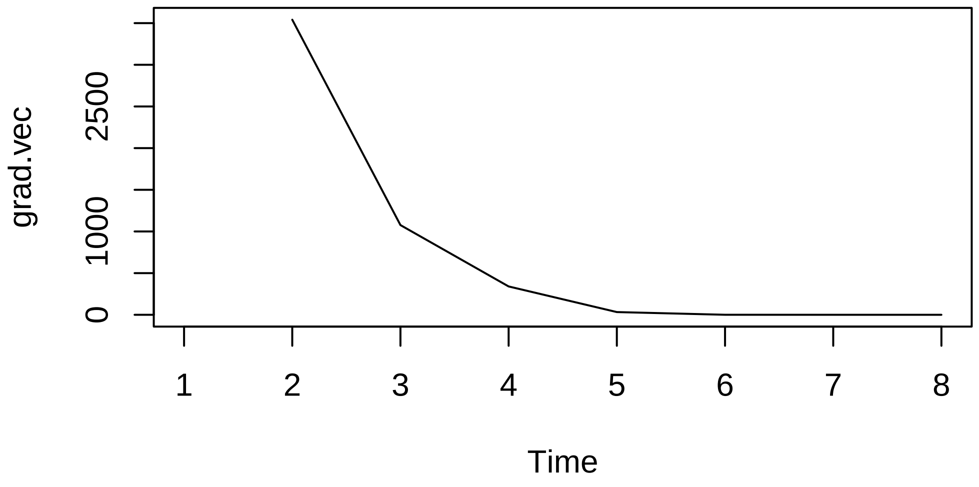
```
    beta.current <- beta.new
    compare <- grad.vec[iter]
  }
  iter
```

```
[1] 8
```

```
beta.new # N-R last iterate.
```

```
                     [,1]
X.Intercept.  1.55679868
Sexmale      -2.52454767
Age          -0.02171550
SibSp        -0.40632009
Parch        -0.22953590
Fare          0.01711193
```

```
plot.ts(grad.vec)
```



4. **In James Cameron's 1997 directorial venture, "Titanic", it is often debated that the character of Jack Dawson (played boisterously by Leonardio DiCaprio) could have probably survived if Rose Bukater (played by the ever-brilliant Kate Winslet) had made room for him on the raft. Let's end this debate once and for all.**

   Jack Dawson was 20 years old when the tragedy happened. He boarded the ship with no family or spouse and paid 7.5 British pounds for his ticket. Rose Bukater was 19 years old on the tragic day. She boarded the ship with her fianceé (treat this as a spouse) and her mother. She paid 512 British pounds for her ticket.

   Using the Newton-Raphson MLE estimates from below, calculate the estimated probabilities of survival for Jack Dawson and Rose Bukater?

(You can answer this question even if you haven't seen Titanic. However, if that is true, you must question all of your life's decisions leading up to this moment).

This may seem like a silly question, but we have to remember that we're statisticians, and not just coders. So after implementing the algorithm, we need to know what to do with the estimate of β. Note that the model itself was:

$$\Pr(Y_i = 1) = \frac{e^{x_i^T\beta}}{1 + e^{x_i^T\beta}} .$$

Having obtained an estimator of β, which is the MLE estimator, we can use that estimator to obtain the probability of success for any ith individual. Here, our final MLE estimator is

```
beta.new
```

```
                   [,1]
X.Intercept.  1.55679868
Sexmale      -2.52454767
Age          -0.02171550
SibSp        -0.40632009
Parch        -0.22953590
Fare          0.01711193
```

Given any x_{new} for an individual, let p_{new} denote the probability of them surviving. Then a reasonable estimator of p_{new} is

$$\hat{p}_{new} = \frac{e^{x_{new}^T\hat{\beta}_{MLE}}}{1 + e^{x_{new}^T\hat{\beta}_{MLE}}} .$$

So from the film, we know information about Jack Dawson and Rose Bukater. The matrix X is has columns in the order of intercept, sex (male = 1), age, SibSp, Parch, Fare. So let's create x_{new} vectors for Rose and Jack:

```
# 1 for intercept, 1 for male,
jack.x <- c(1, 1, 20, 0, 0, 7.5)
rose.x <- c(1, 0, 19, 1, 1, 512)

# estimate from logistic reg is in beta.new
pi.jack <- 1/ (1 + exp( - sum(jack.x * beta.new)))
pi.rose <- 1/ (1 + exp( - sum(rose.x * beta.new)))

pi.jack
```

```
[1] 0.2186212
```

```
pi.rose
```

```
[1] 0.9999058
```

Thus Rose's probability of survival is *very* close to 1, and Jack's probability of survival was around 20%. This is fairly low, and thus, Jack and Rose's love story was doomed from the very beginning.

5. **Implement Gradient Ascent algorithm for the above titanic dataset for logistic regression.**

Since we've already implemented Newton-Raphson, we have the gradient function already written. Thus, implementing Gradient Ascent now requires very little work. However, as we will see, tuning the algorithm to find a good value for the learning rate t, will be the main challenge.

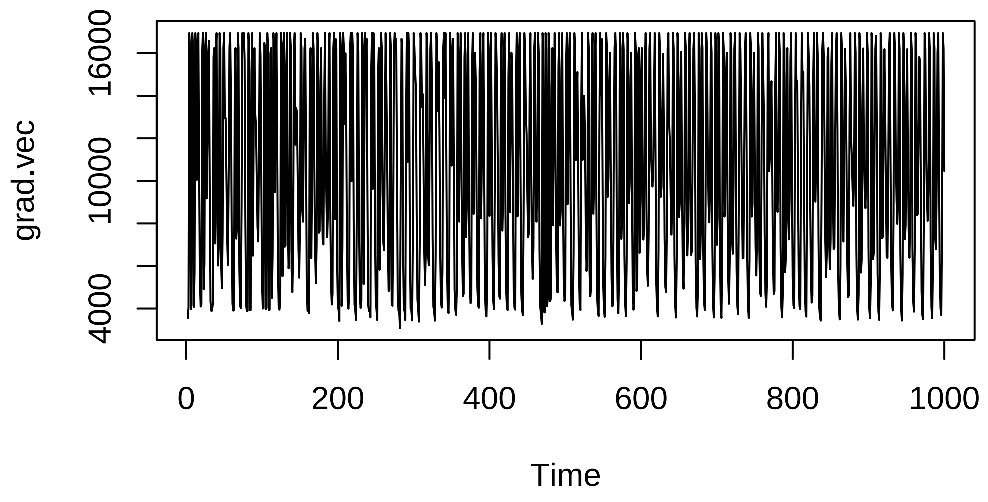
```r
tol <- 1e-10
compare <- 100
iter <- 1

# starting from the zero-vector
grad.vec <- c() # will store gradients here
beta.current <- rep(0, p)
beta.new <- beta.current
t <- 1
while(compare > tol && iter < 1000)
{
  iter <- iter + 1  # tracking iterations

  gradient <- f.gradient(y, X, beta.current)
  beta.new <- beta.current + t * gradient

  grad.vec[iter] <- norm(gradient, "2")
  beta.current <- beta.new
  compare <- grad.vec[iter]
}
iter
```

```
[1] 1000
```

```r
beta.new # GA last iterate.
```

```
X.Intercept.      Sexmale          Age        SibSp        Parch         Fare
    760.2226  -61284.4472   -2313.0212  -37927.0673  -12062.7978   10080.3052
```

```r
plot.ts(grad.vec)
```



In the above the learning rate is clearly too large, since $\|\nabla l(\beta)\|$ oscillated the whole time. When this happens, it typically means that the learning rate t is too large. Let us now reduce it and see what happens:

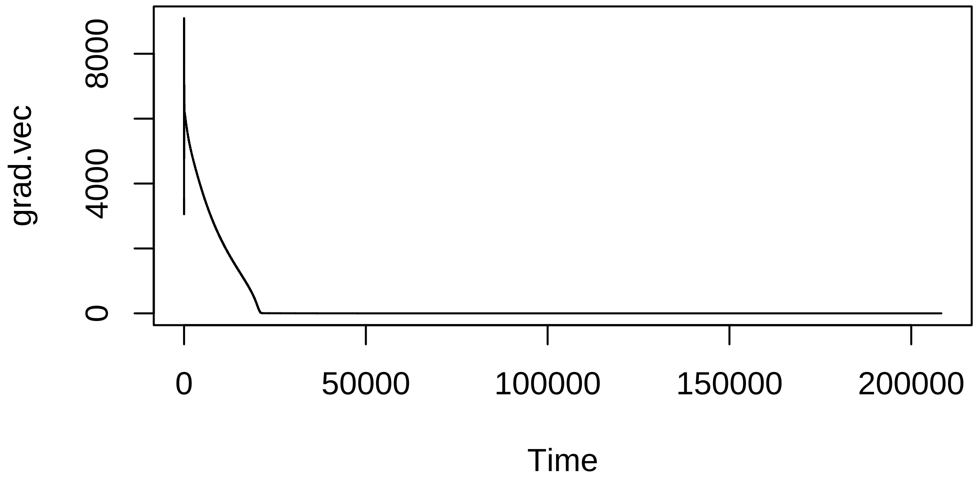
```r
tol <- 1e-5   # reduce the tolerance
compare <- 100
iter <- 1

# starting from the zero-vector
grad.vec <- c() # will store gradients here
beta.current <- rep(0, p)
beta.new <- beta.current
t <- .000007

# increased the max iterations
while(compare > tol && iter < 3e5)
{
  iter <- iter + 1  # tracking iterations

  gradient <- f.gradient(y, X, beta.current)
  beta.new <- beta.current + t * gradient

  grad.vec[iter] <- norm(gradient, "2")
  beta.current <- beta.new
  compare <- grad.vec[iter]
}
iter
```

```
[1] 208303
```

```
beta.new # GA last iterate.
```

```
X.Intercept.      Sexmale          Age        SibSp        Parch         Fare 
  1.55679780  -2.52454720  -0.02171549  -0.40631995  -0.22953579   0.01711193 
```

```
plot.ts(grad.vec)
```



It took a lot of effort to tune this and find a reasonable t. The issue is that component of β are in different scales, and thus the same value of t is inappropriate for different components. There are two ways to tackle this problem: (i) we may center and scale the Xs or (ii) we may come up with different learning rates for each of the components. We may try to do this in another Lab assignment, but for now, we let the above be ok, where I've allowed for 3e5 iterations and even reduced the tolerance.

Alternatively, we can also try changing the learning rate as function of the iterations, k. That is:

$$\beta_{(k+1)} = \beta_{(k)} + t_k \nabla f(\beta_{(k)})$$

There are different ways of choosing t_k, but a common choice is $t_k = tk^{-.5}$. You may try this to improve the convergence rate in any application.

6. **Consider the `motorins` dataset in library `faraway` in R. Do `?motorins` to learn about the dataset.**

```
library(faraway)
?motorins
```

Consider response variables `Claims` in this datasets, which has the number of insurance claims. Using Exercise 12 in 7.3 of the notes, implement the Poisson regression MLE estimation for this dataset. Use both Newton-Raphson and gradient ascent.

In this problem, we have not done Poisson regression in class. However, the mechanism is essentially exactly the same as logistic regression. So let's setup the problem first. We are given that

$$Y_i \sim \text{Poisson}(p_i = e^{x_i^T \beta}) \,.$$

Using this, we can write down the likelihood:

$$L(\beta|y) = \prod_{i=1}^{n} \frac{p_i^{y_i} \exp(-p_i)}{y_i!}$$

This also gives us the log-likelihood:

$$l(\beta) = \sum_{i=1}^{n} y_i \log(p_i) - \sum_{i=1}^{n} p_i - \text{const} = \sum_{i=1}^{n} y_i x_i^T \beta - \sum_{i=1}^{n} e^{x_i^T \beta} - \text{const}$$

Taking derivate to find the gradient, we get:

$$\nabla l(\beta) = \sum_{i=1}^{n} y_i x_i - \sum_{i=1}^{n} x_i e^{x_i^T \beta}$$

From here we can also calculate the Hessian:

$$\nabla^2 l(\beta) = -\sum_{i=1}^{n} x_i x_i^T e^{x_i^T \beta} = -XVX$$

where V is the diagonal matrix with entries $e^{x_i^T \beta}$ in the diagonal. Since the entries of V are positive, the Hessian is negative semi-definite, and we know that our objective function is concave! We can now setup the functions to calculate the Hessian and the Gradient.

```r
f.gradient <- function(y, X, beta)
{
  beta <- matrix(beta, ncol = 1)
  n <- length(y)
  rtn <- colSums( t(y - exp(X %*% beta)) %*% X)
  return(rtn)
}

f.hessian <- function(y, X, beta)
{
  beta <- matrix(beta, ncol = 1)
  V_i <- exp(X %*% beta)
  V <- diag(as.numeric(V_i))
  rtn <- -t(X) %*% V %*% X
  return(rtn)
}
```

Now, let's come to the dataset. Unfortunately, I missed the fact that the first columns of motorins were factor variables. Which means that their treatment and how they make the matrix X will be different. These details are not to be discussed in this course, so instead of using this dataset, I will generate my own data.

```r
set.seed(1)
n <- 50
p <- 5

# generate covariates
X <- cbind(1, matrix( rnorm(n*(p-1)), ncol = p-1, nrow = n))

beta.star <- c(.1, .2, .1, .6, -.3)
pi <- exp(X %*% beta.star)
y <- rpois(n, pi)  # generate response
```

For thus, y, and X, let's try to implement now the Poisson regression estimation using Newton-Raphson.

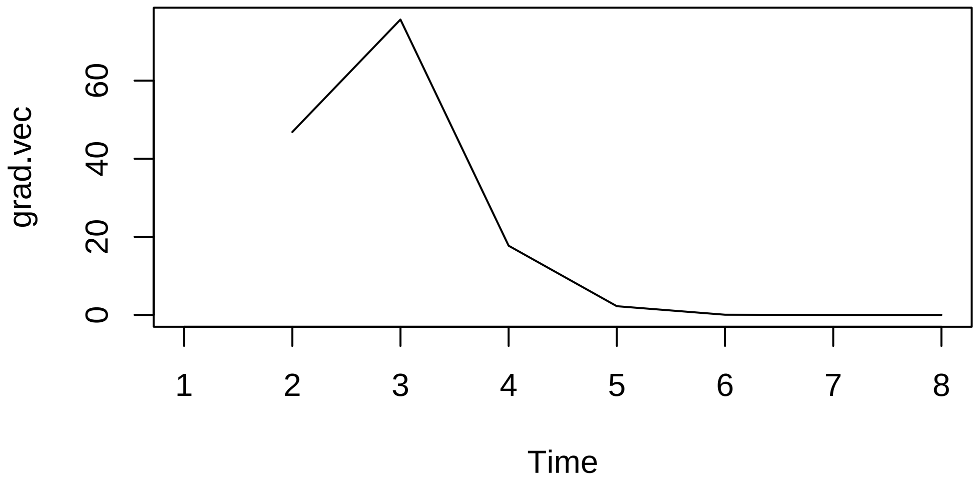
```r
tol <- 1e-10
compare <- 100
iter <- 1

# starting from the zero-vector
grad.vec <- c() # will store gradients here
foo <- glm(y ~ X - 1, family = poisson)
beta.current <- rep(0, p)
beta.new <- beta.current
while(compare > tol)
{
  iter <- iter + 1  # tracking iterations

  gradient <- f.gradient(y, X, beta.current)
  hessian <- f.hessian(y, X, beta.current)
  beta.new <- beta.current - qr.solve(hessian) %*% gradient

  grad.vec[iter] <- norm(gradient, "2")
  beta.current <- beta.new
  compare <- grad.vec[iter]
}

    iter
```

```
[1] 8
```

```
beta.new # N-R last iterate.
```

```
           [,1]
[1,]  0.1996259
[2,]  0.4175556
[3,]  0.2233601
[4,]  0.6147272
[5,] -0.1979004
```

```
plot.ts(grad.vec)
```



We see that the above algorithm converges to a stable value in very few steps. Let's now implement Gradient Ascent algorithm.

```
tol <- 1e-8
compare <- 100
iter <- 1

# starting from the zero-vector
grad.vec <- c() # will store gradients here
beta.current <- rep(0, p)
beta.new <- beta.current
t <- .01
while(compare > tol && iter < 1e5)
{
  iter <- iter + 1  # tracking iterations

  gradient <- f.gradient(y, X, beta.current)
  beta.new <- beta.current + t * gradient
```

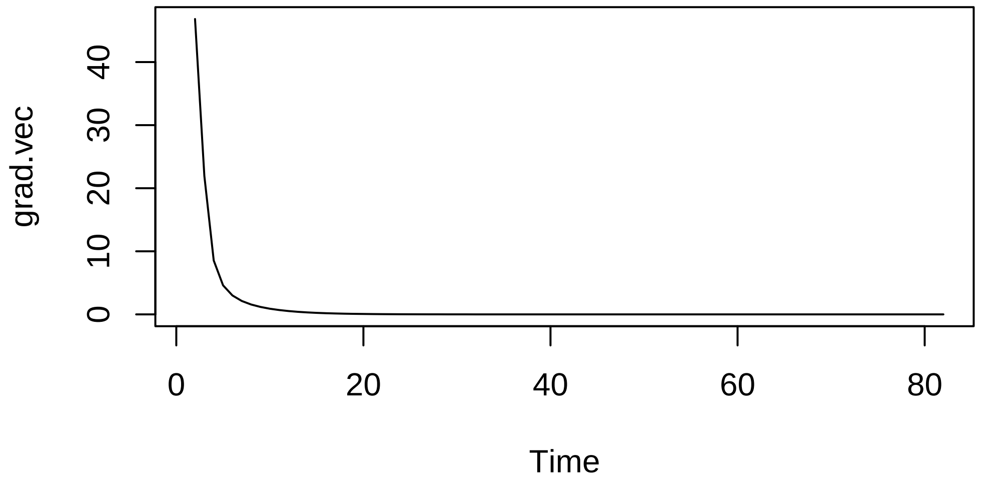
```
    grad.vec[iter] <- norm(gradient, "2")
    beta.current <- beta.new
    compare <- grad.vec[iter]
  }
  iter
```

```
[1] 82
```

```
beta.new # GA last iterate.
```

```
[1]  0.1996259  0.4175556  0.2233601  0.6147272 -0.1979004
```

```
plot.ts(grad.vec)
```



Here again it took some time to tune the gradient ascent algorithm. Try increasing the value of t and see for yourself what happens.