1 c.) For the ill-conditioned case, we now introduce a ridge penalty on the coefficents β. Our optimization problem now becomes

$$\min_{\beta} f(\beta) + \lambda||\beta||^2.$$

As before, we take $f(\beta) = -l(\beta)$, where l is the log-likelihood function of β. Note that

$$\frac{\partial}{\partial \beta_j}||\beta||^2 = 2\beta_j, \quad j = 1, 2, \ldots, p.$$

Thus, the gradient for our steepest descent algorithm is

$$\nabla(f(\beta) + \lambda||\beta||^2) = -\nabla l(\beta) + 2\lambda\beta = -X^T(y - \eta) + 2\lambda\beta.$$

We consider the cases where $\lambda = 0.5, 1, 5$, and run algorithm until $|f(\beta^k) - f(\beta^{k+1}| < 0.001$. For $\lambda = 0.5$, we use a constant step size of $\alpha = .005$.

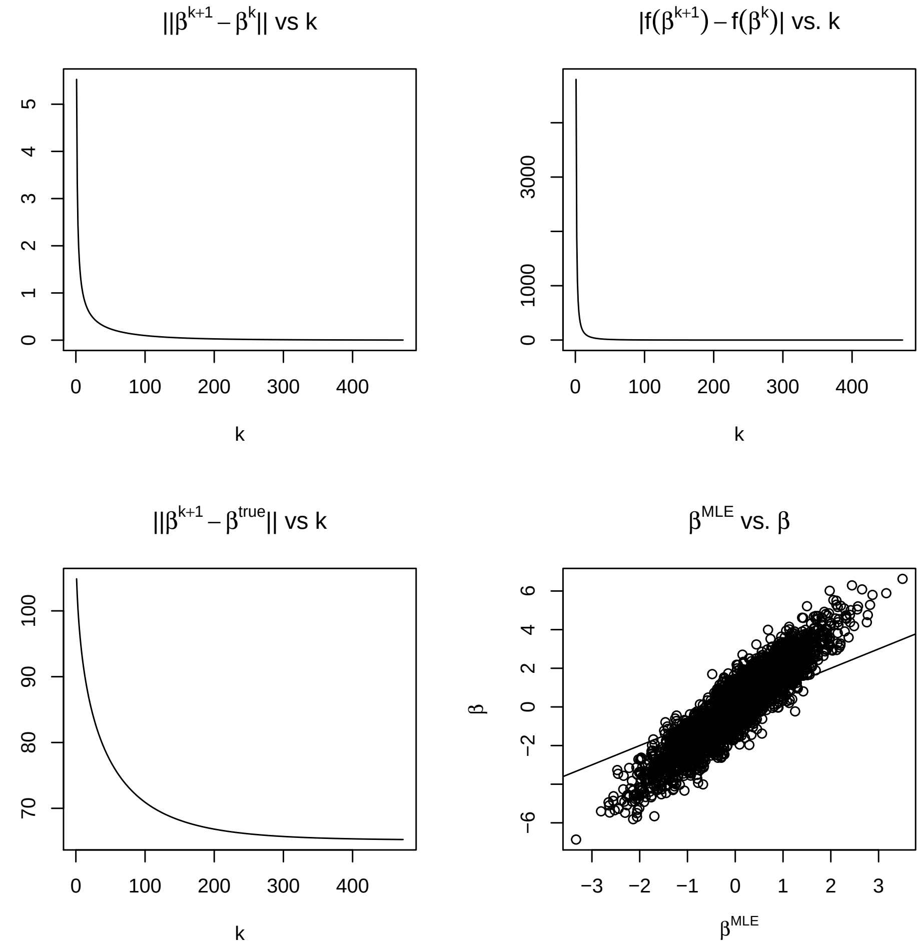
```
> obj.fun <- function(y,p,beta,lambda){
+   -sum(y*log(p)+(1-y)*log(1-p)) + lambda*crossprod(beta,beta)
+ }
> grad.obj <- function(y,X,p,beta,lambda){
+   -crossprod(X,(y-p)) + 2*lambda*beta
+ }
> lambda <- .5
> # Constant step size
> alpha <- .005
> beta.old <- rep(0,p)
> delta <- .001
> maxIter <- 1000
> d.beta <- rep(0,maxIter)
> d.f <- rep(0,maxIter)
> err.beta <- rep(0,maxIter)
> iter <- 1
> eps <- 1
> while(iter<maxIter && eps>delta ){
+   eXb.old <- exp(X%*%beta.old)
+   prob.old <- eXb.old/(1+eXb.old)
+   beta.new <- beta.old - alpha*grad.obj(y,X,prob.old,beta.old,lambda)
+   eXb.new <- exp(X%*%beta.new)
+   prob.new <- eXb.new/(1+eXb.new)
+   eps <- abs(obj.fun(y,prob.new,beta.new,lambda)-obj.fun(y,prob.old,beta.old,lambda))
+   d.f[iter] <- eps
+   d.beta[iter] <- sqrt(crossprod((beta.new-beta.old),(beta.new-beta.old)))
+   err.beta[iter] <- sqrt(crossprod((beta.new-true.beta),(beta.new-true.beta)))
+   beta.old <- beta.new
+   iter <- iter+1
+ }

> # Iterations
> iter

[1] 473

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))
```

```
          [,1]
[1,] 65.25788
```



For the decreasing step size we use $\alpha^k = 0.5/\sqrt{k}$.

```
> alpha0 <- .05
> alpha <- alpha0
> beta.old <- rep(0,p)
> delta <- .001
> maxIter <- 1000
> d.beta <- rep(0,maxIter)
> d.f <- rep(0,maxIter)
> err.beta <- rep(0,maxIter)
> iter <- 1
> eps <- 1
> while(iter<maxIter && eps>delta ){
+   eXb.old <- exp(X%*%beta.old)
+   prob.old <- eXb.old/(1+eXb.old)
+   beta.new <- beta.old - alpha*grad.obj(y,X,prob.old,beta.old,lambda)
+   eXb.new <- exp(X%*%beta.new)
```

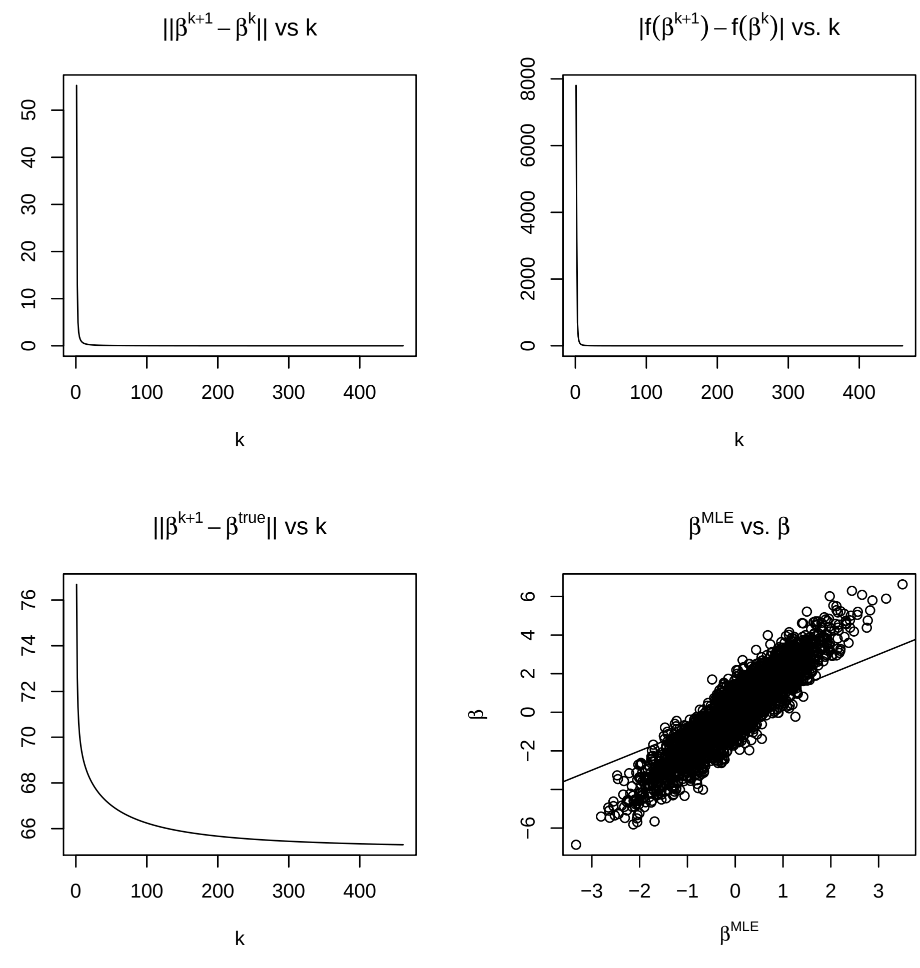
```
+   prob.new <- eXb.new/(1+eXb.new)
+   eps <- abs(obj.fun(y,prob.new,beta.new,lambda)-obj.fun(y,prob.old,beta.old,lambda))
+   d.f[iter] <- eps
+   d.beta[iter] <- sqrt(crossprod((beta.new-beta.old),(beta.new-beta.old)))
+   err.beta[iter] <- sqrt(crossprod((beta.new-true.beta),(beta.new-true.beta)))
+   beta.old <- beta.new
+   iter <- iter+1
+   alpha <- alpha0/sqrt(iter)
+ }

> # Iterations
> iter

[1] 461

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

        [,1]
[1,] 65.2969
```



For Armijo's method we use $s = 0.01$, $\sigma = 0.9$ and $\gamma = 0.8$.

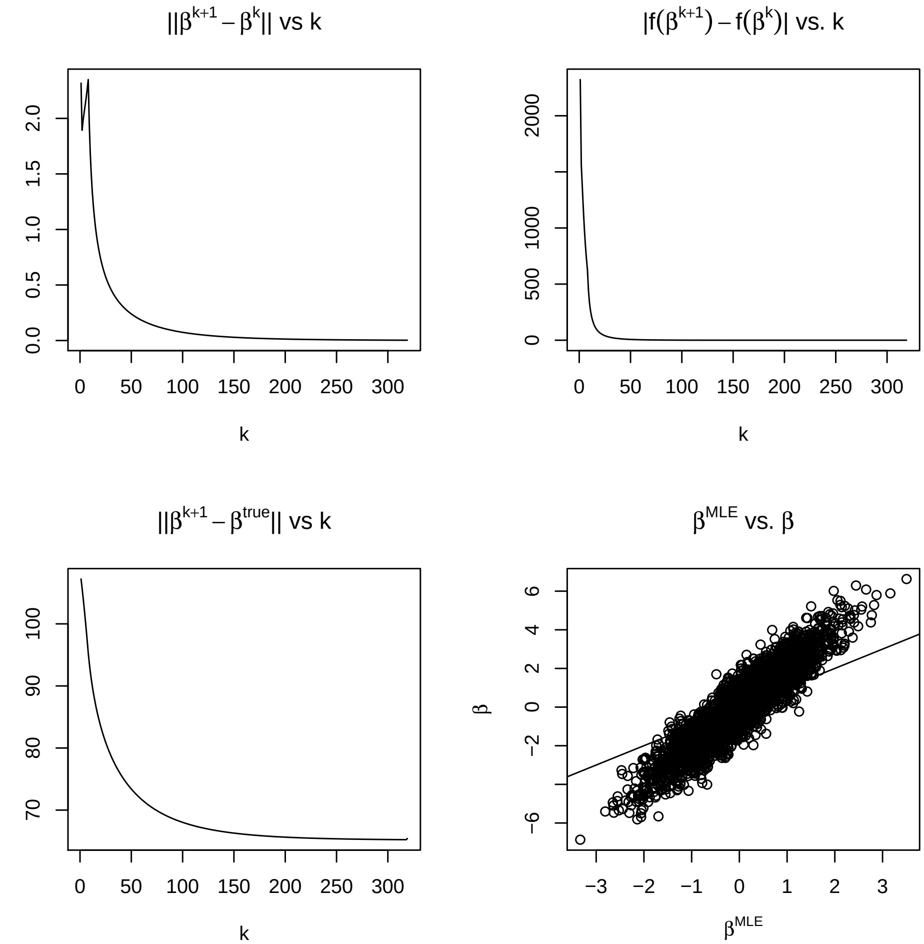
```
> s <- .01
> sigma <- .9
> gamma <- .8
> beta.old <- rep(0,p)
> delta <- .001
> maxIter <- 1000
> d.beta <- rep(0,maxIter)
> d.f <- rep(0,maxIter)
> err.beta <- rep(0,maxIter)
> iter <- 1
> eps <- 1
> while(iter<maxIter && eps>delta ){
+   check <- 0
+   t <- 1
+   while(check==0){
+     eXb.old <- exp(X%*%beta.old)
+     prob.old <- eXb.old/(1+eXb.old)
+     grad.f <- grad.obj(y,X,prob.old,beta.old,lambda)
+     beta.new <- beta.old - s*(gamma^t)*grad.f
+     eXb.new <- exp(X%*%beta.new)
+     prob.new <- eXb.new/(1+eXb.new)
+     a <- obj.fun(y,prob.old,beta.old,lambda)-obj.fun(y,prob.new,beta.new,lambda)
+     b <- sigma*s*(gamma^t)*t(grad.f)%*%grad.f
+     if(a >= b){
+       check <- 1
+     }
+     else{t <- t+1}
+   }
+   eps <- abs(obj.fun(y,prob.new,beta.new,lambda)-obj.fun(y,prob.old,beta.old,lambda))
+   d.f[iter] <- eps
+   d.beta[iter] <- sqrt(crossprod((beta.new-beta.old),(beta.new-beta.old)))
+   err.beta[iter] <- sqrt(crossprod((beta.new-true.beta),(beta.new-true.beta)))
+   beta.old <- beta.new
+   iter <- iter+1
+ }

> # Iterations
> iter

[1] 319

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

         [,1]
[1,] 65.23012
```



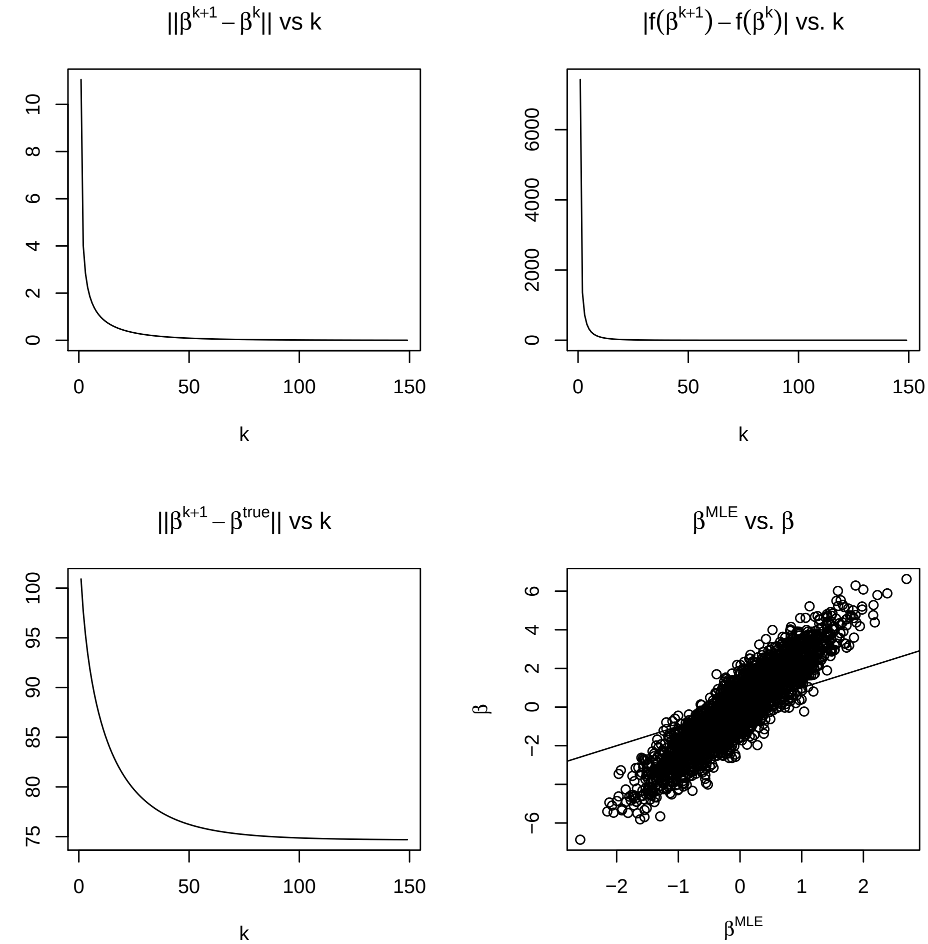
For $\lambda = 1$, we used a constant step size $\alpha = 1$

```
> # Iterations
> iter

[1] 149

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

          [,1]
[1,] 74.69017
```



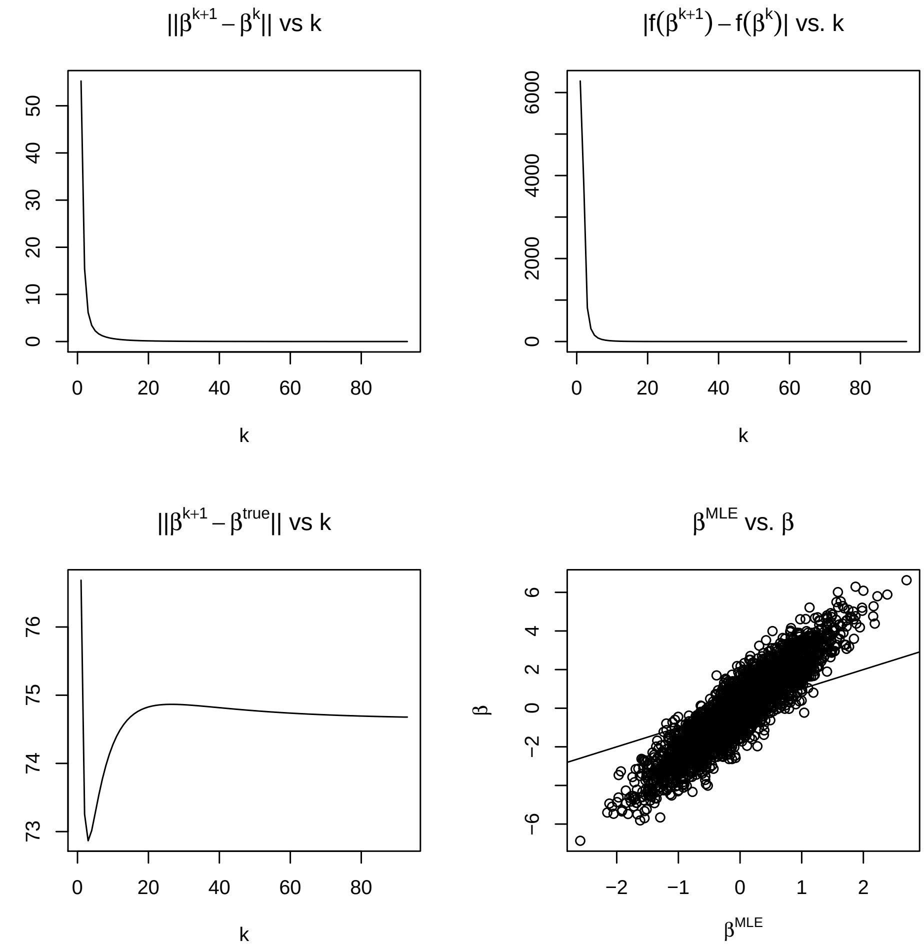
For the decreasing step size we use $\alpha^k = 0.5/\sqrt[3]{k}$

```
> # Iterations
> iter

[1] 93

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

         [,1]
[1,] 74.67936
```



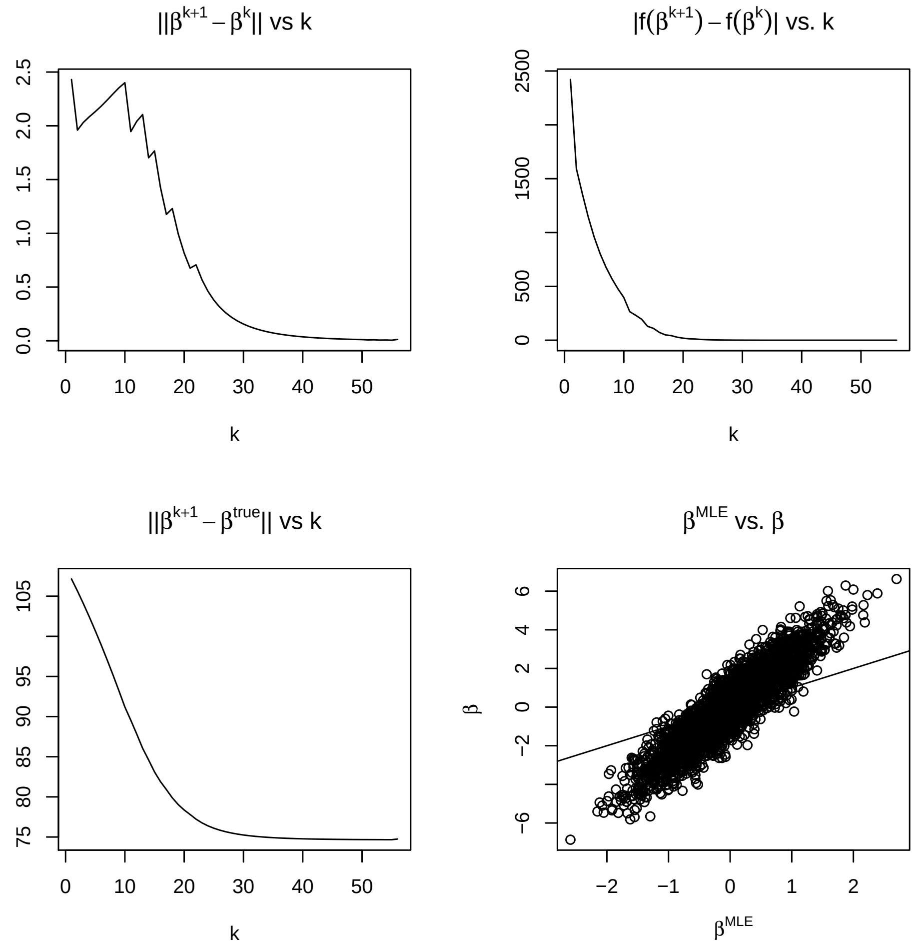
For Armijo's method we used,

```
> # Iterations
> iter
```

```
[1] 56
```

```
> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))
```

```
          [,1]
[1,] 74.66557
```



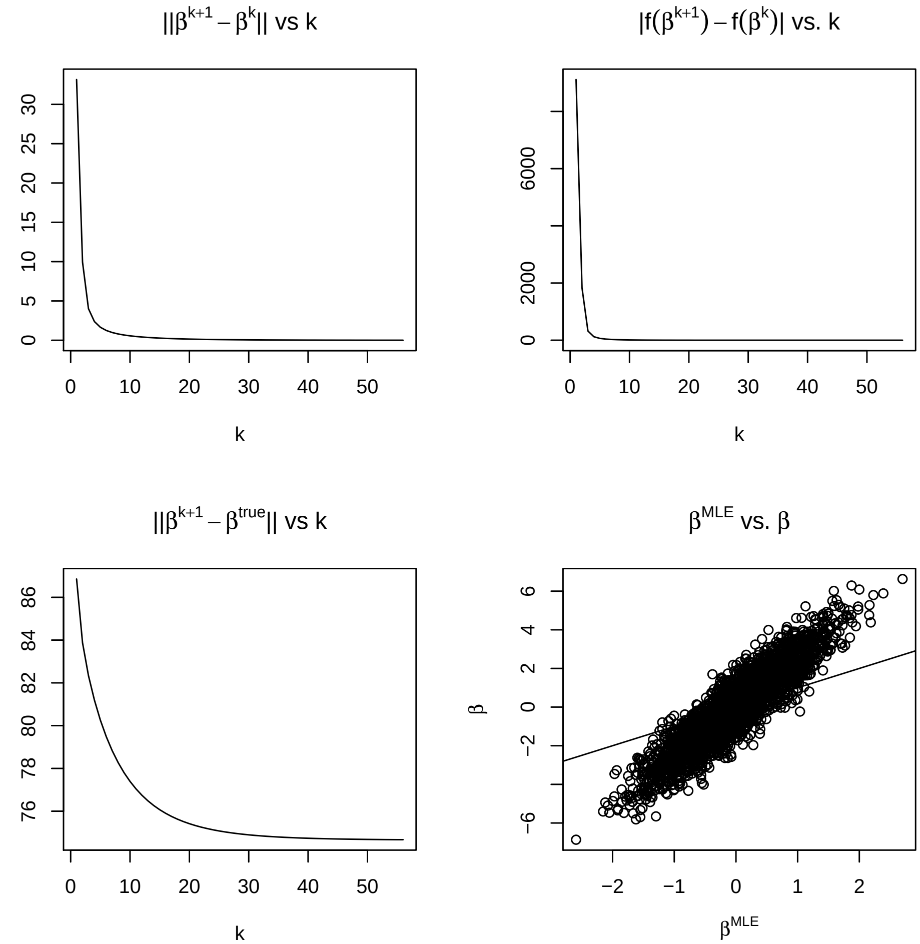
For $\lambda = 5$, we use a step size of $\alpha = 0.03$

```
> # Iterations
> iter
```

```
[1] 1000
```

```
> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))
```

```
         [,1]
[1,] 89.00651
```



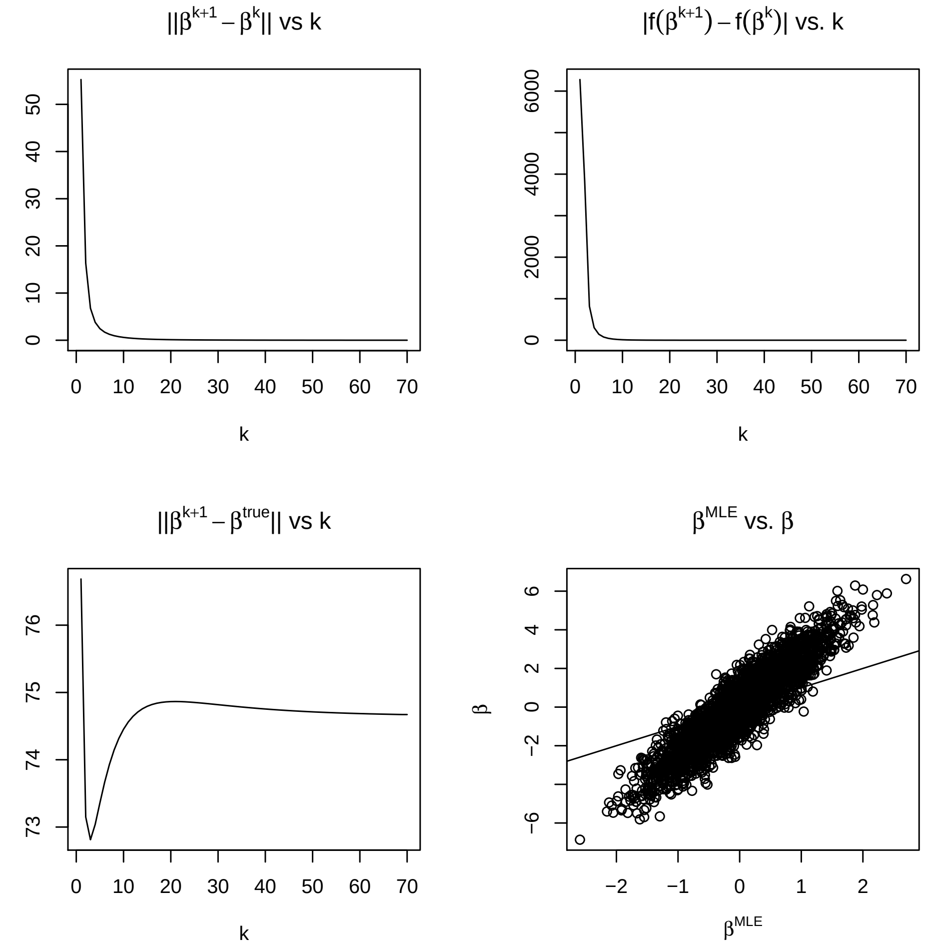
For the decreasing step algorithm, we chose a step-size of $\alpha^k = 0.05/\sqrt[4]{k}$.

```
> # Iterations
> iter

[1] 47

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

         [,1]
[1,] 91.20833
```



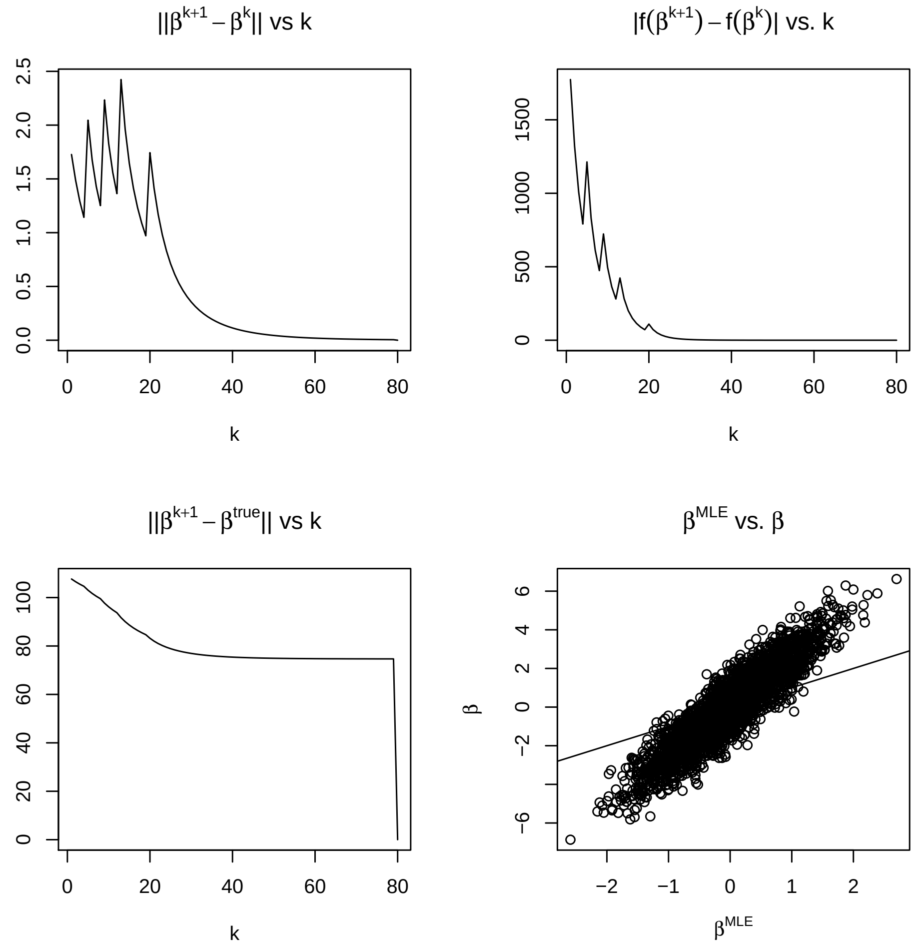
For Armijo's rule, we took $s = 0.05$, $\sigma = 0.9$ and $\gamma = 0.5$.

```
> # Iterations
> iter

[1] 57

> # Error
> sqrt(t(true.beta-beta.old)%*%(true.beta-beta.old))

         [,1]
[1,] 91.21664
```



Larger values of λ improve the convergence of the algorithms since increasing λ improves the condition number. However, increasing λ also results in poorer estimates since the vaules of β^{MLE} are being shrunk to zero as λ gets large.