Implementations of Optimisation Algorithms in R with Applications to the Rosenbrock Function

We refer to Amir Beck's "Introduction to Nonlinear Optimization - Theory, Algorithms and Applications"

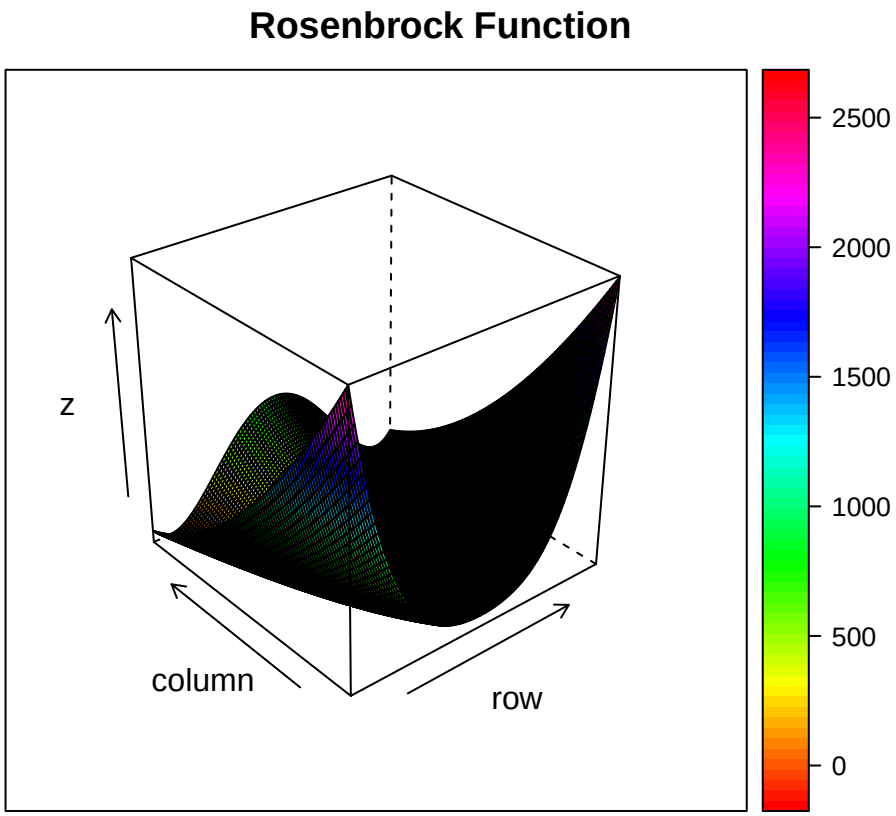
We demonstrate implementations of different optimisation algorithms, namely:

- Gradient Method
- Hybrid Newton Method
- Simulated Annealing

These are incorporated into a package, `myOptPackage`.

We benchmark each algorithm's performance on the **Rosenbrock function**, notorious in numerical optimisation for being very difficult to minimise algorithmically.

```
#install.packages("lattice")
library(lattice)
a <- 1; b<- 100 #set fixed parameters as standard for Rosenbrock function

f <- function(x,y) (a-x)^2 + b*(y-x^2)^2 #define Rosenbrock function

coords <- data.frame(x =seq(-2, 2, length=101), #store grid
       y =seq(-1, 3, length=101))

z <- outer(coords[,1], coords[,2], f) #apply function

wireframe(z, drape = T, col.regions=rainbow(3000), main = "Rosenbrock Function" ) #show contour plot
```



Gradient Method

Based on Beck, example 4.6, p.54

Implementation of the **Gradient Method**, or **Gradient Descent**. We iterate methods on

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + t_k \boldsymbol{d}_k \qquad\qquad k = 0, 1, 2, ...$$

where the descent direction $\boldsymbol{d}_k$ is the negative gradient $-\nabla f(\boldsymbol{x}_k)$, and the stepsize t_k is chosen with backtracking. To do this, we require the first derivative of the Rosenbrock function.

```
f1 <- deriv(expression((a-x)^2 + b*(y-x^2)^2), namevec = c('x', 'y'), function.arg=T, hessian=F) #calcu

GradDesc <- function(f1,x0, s=1, alpha = 1/4, beta = 1/2, eps = 1e-2, max_iter = 50000){
  x <- x0 #set initial coordinate
  iter <- 0 #set iterations to 0
  grad <-  attr(f1(x[1], x[2]), "gradient") #compute gradient at current coordinate
  fun_val <- rep(0, max_iter) #initiate vector of function values
  coords <- matrix(NA, nrow =  max_iter, ncol = 2) #initiate vector of coordinates


  while (norm(grad, type ="2") > eps && iter < max_iter){
    iter <- iter + 1 #increment iterations
    t <- s #set t to initial stepsize
    change <- x - t* grad #set change in coords

    while (f1(x[1],x[2]) - f1(change[1],change[2]) < alpha * t * norm(grad, type ="2")^2) {
       t <- beta * t  #set stepsize by backtracking
       change <- x - t* grad #set change in coords
    }

    x <- x-t*grad #move coords
    grad <-  attr(f1(x[1], x[2]), "gradient") #compute gradient at current coordinate
    fun_val[iter] <- f1(x[1], x[2]) #compute value of function at current coordinate
    #print(fun_val[iter])
    coords[iter,] <- t(x) #store coordinates
  }


  results <- c(Iterations = iter, #print step
    Coordinates = coords[iter,], #print optima coordinates
    Gradient = grad, #print gradient
    Value = fun_val[iter]) # print fn val
  attr(results, "coords") <- coords #set ordered coordinates as attribute
  attr(results, "fun_val") <- fun_val #set ordered function values as attribute
  return(results)
}
```

Run for the Rosenbrock function, starting at (10,10)

```
system.time(GD_Rosen <- GradDesc(f1, x0 = c(10,10), eps = 1e-2))
```

```
##    user  system elapsed
##  11.147   0.000  11.149
```

```
GD_Rosen[1:6]
```

```
##   Iterations Coordinates1 Coordinates2    Gradient1    Gradient2
## 2.329500e+04 1.010433e+00 1.021012e+00 6.251250e-03 7.232218e-03
##        Value
## 1.089845e-04
```

This finds an approximate minimum in 23295 steps, and takes 10.296 seconds to run.

Newton Method (Hybrid)

Based on Beck, p.93

An alternative approach is **Newton's Method**, which uses the Hessian to update coordinates according to

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - [H(f(\boldsymbol{x}))]^{-1} \nabla f(\boldsymbol{x})$$

This requires that the Hessian matrix H is positive definite at all points in the space, however, while the Rosenbrock function is merely positive semidefinite over $\mathbb{R}^2$. We hence use the **Hybrid Gradient-Newton Method**, which employs the Newton method for points where the Hessian is positive definite, and uses the gradient method otherwise.

```
library(matrixcalc) #load matrixcalc for pos.def. test is.positive.definite
f2 <- deriv(expression((a-x)^2 + b*(y-x^2)^2), namevec = c('x', 'y'), function.arg=T, hessian=T)
#calculate gradient and Hessian


hybrid <- function(f2,x0, s=1, alpha = 1/4, beta = 1/2, eps = 1e-2, max_iter = 1000){
  x <- x0 #set initial coordinate
  iter <- 0 #set iterations to 0
  grad <-  attr(f2(x[1], x[2]), "gradient") #compute gradient at current coordinate
  hess <- matrix( attr( f2(x[1], x[2]), "hessian"), nrow = 2, ncol =2 )
  #compute hessian at current coordinate
  #hval <- hessian(x) #set hessian
  fun_val <- rep(0, max_iter) #initiate vector of function values
  coords <- matrix(NA, nrow =  max_iter, ncol = 2) #initiate vector of coordinates


  if(is.positive.definite(hess)){
    d <- solve(hess, t(grad)) #if gradient is pos.def. use Newton direction
  } else {
        d <- grad #else use gradient direction
      }

  while (norm(grad, type ="2") > eps && iter < max_iter){
    iter <- iter + 1 #increment iterations
    t <- s #set t to initial stepsize
    change <- x - t* d #set change in coords

    while (f2(x[1],x[2]) - f2(change[1],change[2]) < alpha * t * grad %*% d) {
       t <- beta * t  #set stepsize by backtracking
       change <- x - t* d #set change in coords
    }
```

```
    x <- x-t*d #move coords
    grad <-  attr(f1(x[1], x[2]), "gradient") #compute gradient at current coordinate
    hess <- matrix( attr( f2(x[1], x[2]), "hessian"), nrow = 2, ncol =2 )
    #compute hessian at current coordinate
    fun_val[iter] <- f1(x[1], x[2]) #compute value of function at current coordinate
    #print(fun_val[iter])
    coords[iter,] <- t(x) #store coordinates

      if(is.positive.definite(hess)){
      d <- solve(hess, t(grad)) #if gradient is pos.def. use Newton direction
       } else {
        d <- grad #else use gradient direction
      }
  }

  results <- c(Iterations = iter, #print step
    Coordinates = coords[iter,], #print optima coordinates
    Gradient = grad, #print gradient
    Value = fun_val[iter]) # print fn val


  attr(results, "coords") <- coords #set ordered coordinates as attribute
  attr(results, "fun_val") <- fun_val #set ordered function values as attribute

  results
}
```

```
system.time(hybrid_Rosen <- hybrid(f2, x0 = c(0,0), eps = 1e-2))
```

```
##    user  system elapsed
##   0.076   0.000   0.076
```

```
hybrid_Rosen[1:6]
```

```
##    Iterations  Coordinates1  Coordinates2     Gradient1     Gradient2
##  1.300000e+01  9.999990e-01  9.999979e-01  1.389278e-05 -7.976462e-06
##         Value
##  1.220093e-12
```

This takes 0.068 seconds to run.

Simulated Annealing

Simulated Annealing is based on the physical process of annealing, adapting hill-climbing algorithms to avoid getting stuck at local optima. By accepting neigbouring points with lower function values, according to the current temperature, we may overcome this and obtain the global optimum.

The procedure for minimisation follows this algorithm:

- Input the initial coordinates $\boldsymbol{x}_0$, maximum iterations k_{max}, cooling rate $\alpha \in (0, 1)$
- Set
$$\boldsymbol{x} \leftarrow \boldsymbol{x}_0$$
- For
$$k = 1, ...k_{max}$$

- \[T \xleftarrow{} \alpha T \]
- \[\boldsymbol{x}_{new} \xleftarrow{} neighbour(\boldsymbol{x}) \]
- If \[P(E(\boldsymbol{x}), E(\boldsymbol{x}_{new}),T) \geq unif[0,1] \] then \[\boldsymbol{x} \xleftarrow{} \boldsymbol{x}_{new} \]
- Output: $\boldsymbol{x}$

```
library(MASS)

#define simulated annealing function
SimAnneal <- function(f,x0 =  c(0,0), alpha = 0.95, sig = 5, max_iter = 100000){
  x <- matrix(nrow = max_iter+1, ncol = 2) #initialise coordinate matrix
  x[1,] <- t(x0) #initialise x0
  Temp <- 1
  fun_val <- rep(0, max_iter)

  for (k in 1:max_iter) {
    Temp <- alpha * Temp #apply cooling
    delta <- mvrnorm(1, c(0,0) , Sigma = sig*diag(nrow= 2) )
    x_new <- x[k,] +  t(delta)  #perturb location
    dif <- f(x_new[1], x_new[2]) - f(x[k,1], x[k,2])  #calculate function change
    if(f(x_new[1], x_new[2]) < f(x[k,1], x[k,2])){
      x[k+1,] <- x_new #if decreasing, accept
    } else if (runif(1) > exp(dif/Temp) ){
        x[k+1,] <- x_new # accept according to acceptance probability
    } else {
        x[k+1,] <- x[k,]
    }
  fun_val[k] <- f(x[k,1], x[k,2])
  }

  results <-  c(Iterations = k, #print step
    Coordinates = x[k,], #print optima coordinates
    #Gradient = grad, #print gradient
    Value = f(x[k,1], x[k,2]) ) # print fn val

    attr(results, "coords") <- coords #set ordered coordinates as attribute
  attr(results, "fun_val") <- fun_val #set ordered function values as attribute

  results

}
```

```
set.seed(123)
system.time(SA_Rosen <- SimAnneal(f, c(0,0), max_iter = 50000 ))
```

```
##    user  system elapsed
##   3.892   0.008   3.901
```

```
SA_Rosen[1:4]
```

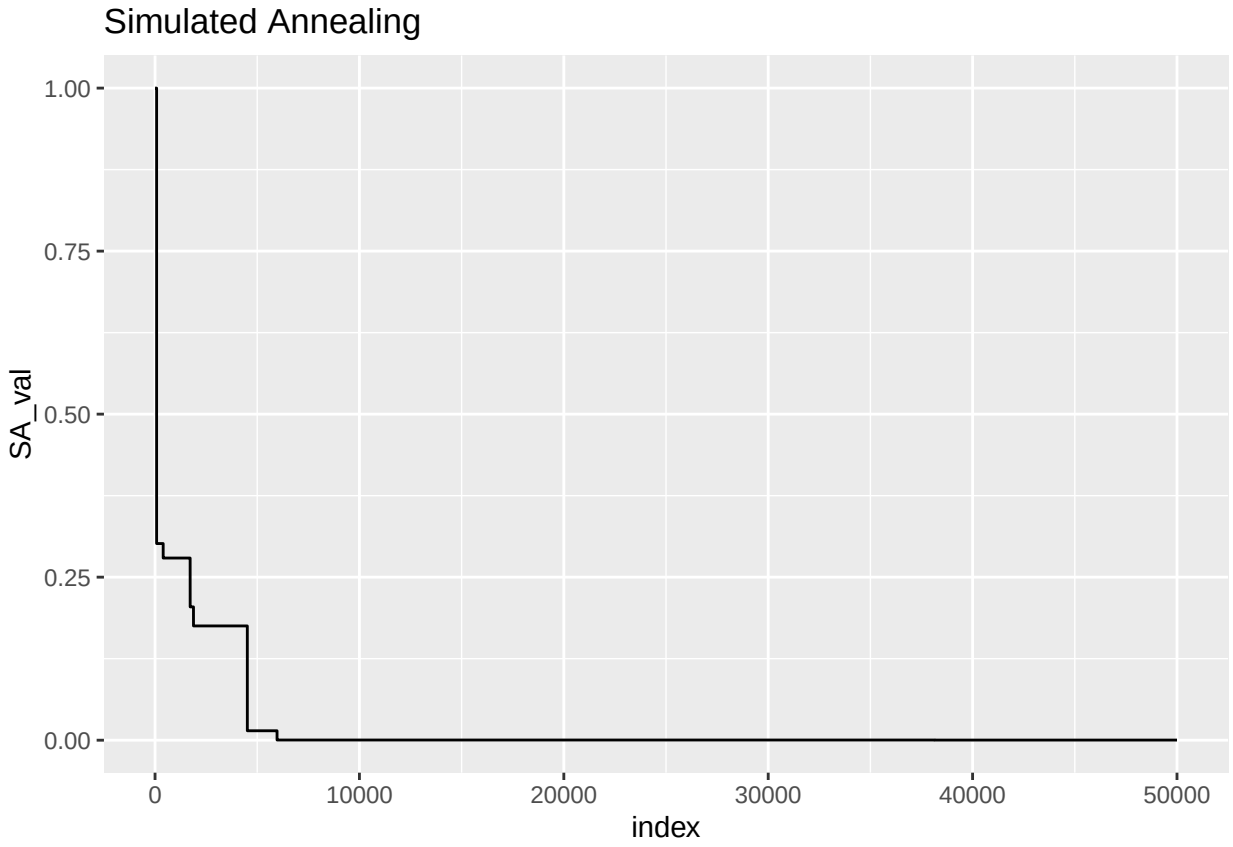
```
##   Iterations Coordinates1 Coordinates2        Value
## 5.000000e+04 1.012978e+00 1.026316e+00 1.721260e-04
```

This takes 3.635 seconds to run.

Comparison

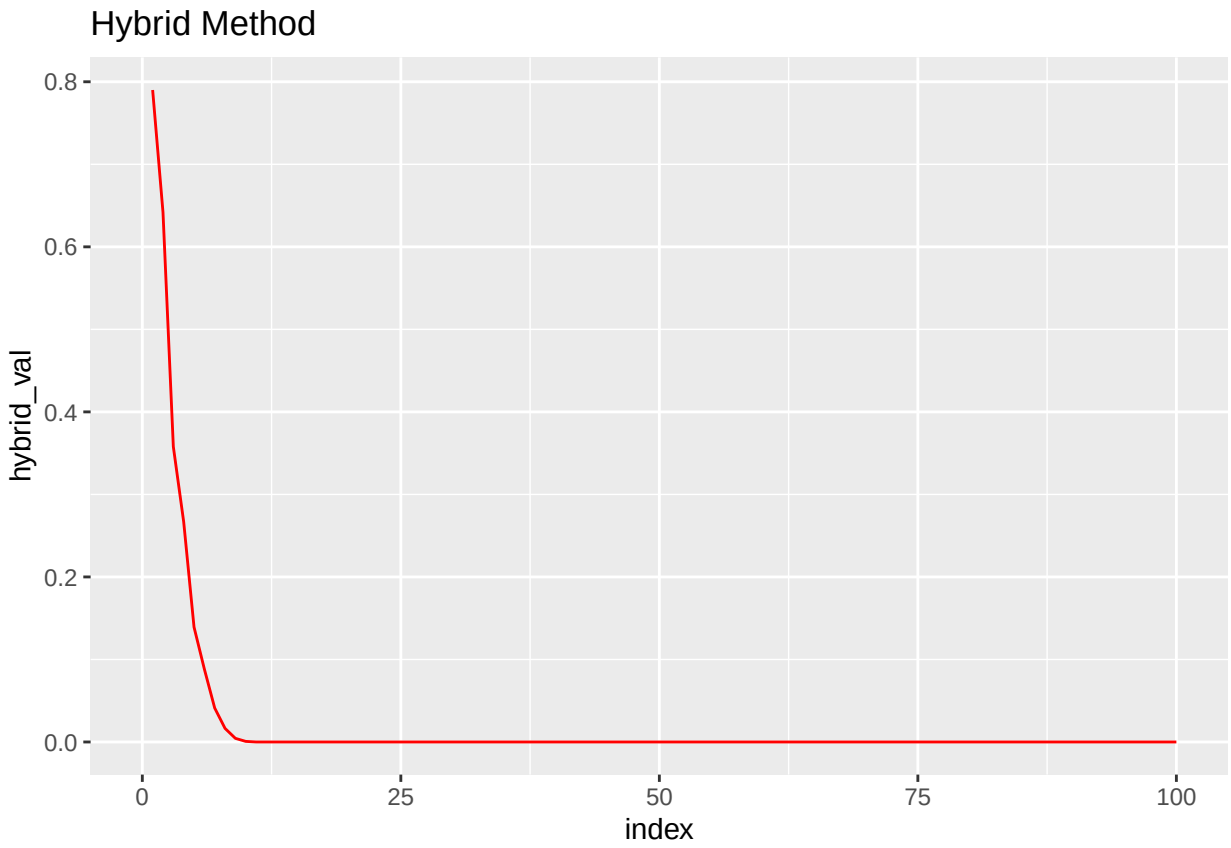
We collect and compare the function value of each method as a function of the iteration.

```
library(ggplot2)
GD_val <- attr(GD_Rosen, "fun_val") #extract function value vectors
hybrid_val <- attr(hybrid_Rosen, "fun_val")
SA_val <- attr(SA_Rosen, "fun_val")

n <- max(length(GD_val), length(hybrid_val), length(SA_val)) # fill non-entries with NA
length(GD_val) <- n
length(hybrid_val) <- n
length(SA_val) <- n

fun_val_df <- data.frame(index = 1:n, GD = GD_val, hybrid = hybrid_val, SA = SA_val) #data.frame(matrix

ggplot(fun_val_df, aes(x= index) ) +  geom_line( aes(y = SA_val)) + ggtitle("Simulated Annealing")
```



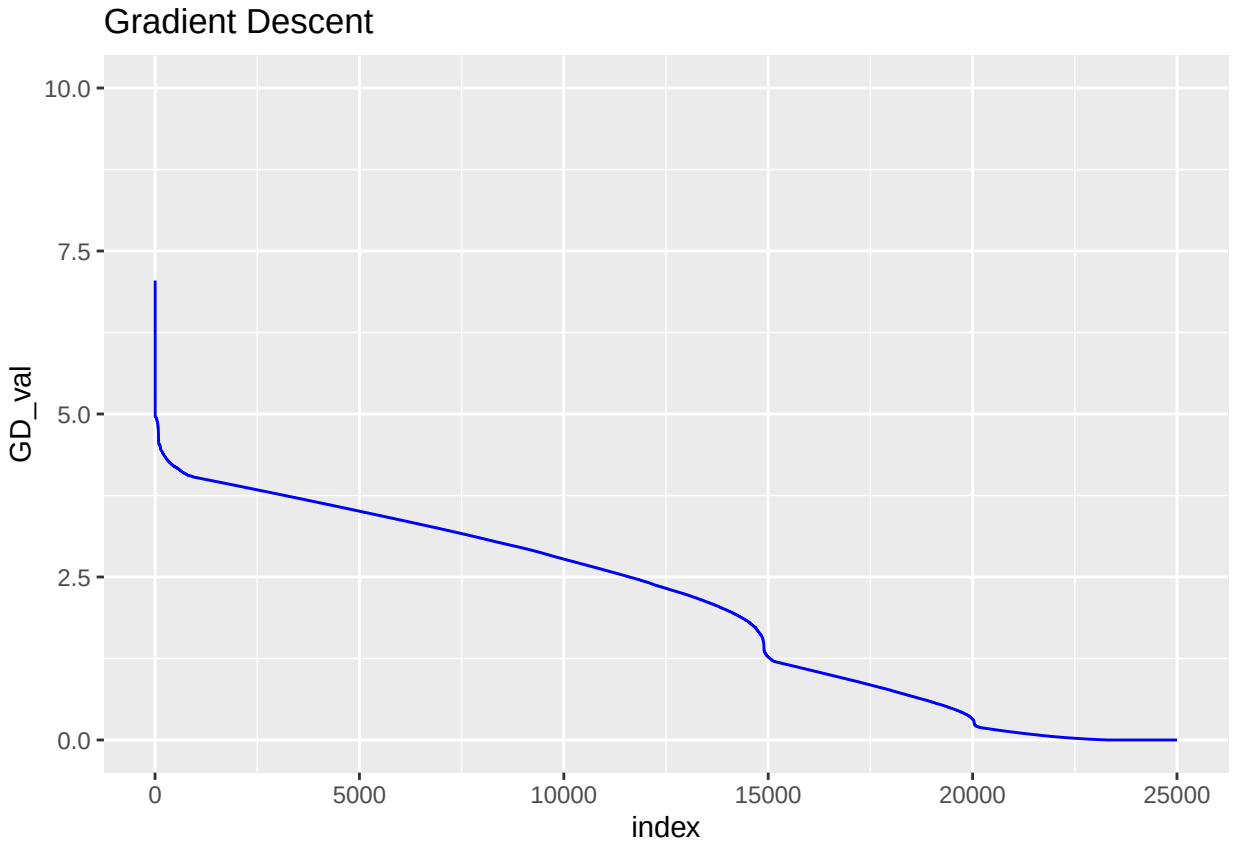

```
ggplot(fun_val_df, aes(x= index) ) +  geom_line( aes(y = hybrid_val), color = "red") + xlim(0, 100) + gg
```

```
## Warning: Removed 49900 rows containing missing values (geom_path).
```




```
ggplot(fun_val_df, aes(x= index) ) +  geom_line( aes(y = GD_val), color = "blue") + xlim(0, 25000) + yl
```

```
## Warning: Removed 25002 rows containing missing values (geom_path).
```



We can conclude that of the three methods we considered, the best performing algorithm for the Rosenbrock function in terms of both real time and iterations is the hybrid method. This is unsurprising, given that second-order methods perform far better in general than first-order or metaheuristic methods.