Optimisation 874

Post Block Assessment 2

Francois van Zyl: 18620426

06 August 2020

DBMOSA implementation

The goal of this assignment is to implement a dominance-based multi-objective simulated annealing (DBMOSA) algorithm to solve a minimisation problem with $f_1(x) = x^2$, and $f_2(x) = (x-2)^2$ with x being an element of $[-10^5, 10^5]$. I worked in RStudio, and started the problem by clearing my workspace and writing a simple function that can evaluate the objective functions for certain values of x. This function takes as input a value or vector of values representing x, and returns a dataframe containing the objective values calculated at this value of x.

```
rm(list = ls()) # Clear workspace
evaluate_objective <- function(x) # Function to evaluate objectives
{
  f1 = x^2  # Defined objective values
  f2 = (x-2)^2
  return(data.frame(f1, f2))  # Return objective values
}
```

Thereafter I wrote a function to generate neighbours for the DBMOSA algorithm. There is no standard procedure to do implement this, however it is required to be random to exploit the fact that DBMOSA can accept non-improving solutions. I decided to implement a neighbourhood function that perturbs the existing solution according to the current and initial temperature settings. At high temperatures, DBMOSA act as a sort of random walk where the algorithm has a higher probability of choosing non-improving moves, and at lower temperatures, the probability of accepting non-improving moves declines. Therefore, I thought it would be interesting to implement a neighbourhood function that generates a random neighbour within predefined limits that are controlled by the ratio of the current temperature to the initial temperature. Therefore, at higher temperatures the function has the possibility to generate neighbours that are farther away, and as the temperature is decreased this distance that the neighbours can be drawn from is decreased. I set up the function such that a high temperatures, the function will generate neighbours that are more likely to be far away, at medium temperatures, the function will generate neighbours that are more likely to be far away but less far than at high temperatures, and at low temperatures, the function will generate neighbours that are closer than the previous temperatures. I defined high temperatures as temperatures that fall above 2/3 of the initial starting temperature, medium temperatures as temperatures that fall between 1/3 and 2/3 of the initial starting temperature, and low temperatures as temperatures that fall below 1/3 of the intial starting temperature. At these three brackets, (high, medium and low) the random perturbation that is applied to the decision value is drawn from a random uniform distribution with limits corresponding to $[-10^4, 10^4]$, $[-10^3, 10^3]$, and $[-10^1, 10^1]$ respectively. The function is displayed below, and it takes as input the current decision value x, the current temperature *temp*, as well as the initial temperature *temp_init*, and returns the perturbed point *x_new*.

```
generate_neighbour <- function(x, temp, temp_init)
{
  limits <- c(-10^5, 10^5) # Limits of decision space
  x_new <- Inf # Ensure while loop is entered
  if(temp > 2*temp_init/3) # If the temp is in high bracket
  {
    while(!(x_new > limits[1] && x_new < limits[2]))  # While x_new is out of limits
    {
      x_new <- x + runif(1, min = -10^4, max = 10^4) # Generate a new random point
    }
  }
  if(temp > temp/3 && temp < 2*temp_init/3 ) # If the temp is in med bracket
  {
    while(!(x_new > limits[1] && x_new < limits[2])) # While x_new is out of limits
    {
      x_new <- x + runif(1, min = -10^2, max = 10^2) # Generate a new random point
    }
  }
  if(temp < temp_init/3) # If the temp is in low bracket
  {
    while(!(x_new > limits[1] && x_new < limits[2])) # While x_new is out of limits
    {
      x_new <- x + runif(1, min = -10^0, max = 10^0) # Generate a new random point
    }
  }
  return(x_new) # Return the perturbed point
}
```

Thereafter I wrote a function that will act to accept a solution into the archive **A**. The function takes as input the existing archive A, and the current decision value x. It then evaluates the objective of x, and binds it into A row-wise (meaning each row in A corresponds to a non-dominated solution) and returns the new A.

```
accept_soln <- function(A, x)
{
  f <- evaluate_objective(x) # Evaluate the objective of x
  return(rbind(A, f)) # Bind
}
```

The following function was used to count how many solutions exist within A that dominate x. The function takes as input the existing archive A, and the decision variable x which represents either the objective function values of the current or neighbouring solution. A counter is initialized after which a for loop is entered, in which a single if statement is computed for each row within the existing archive. This if statement is only passed if the entry within A is at least as good as x in $f_1(x)$ and even better than x in $f_2(x)$, or alternatively if the entry in A is better in $f_1(x)$ than x and at least as good in $f_2(x)$ as x. This is the definition of domination, in which case A dominates x. Therefore if this if statement returns true, the counter is incremented implying that the row or entry within A dominates the objective values of x.

```
num_dom_soln <- function(A, x) # Function to count how many solns within A dominate x
{
  counter <- 0 # Initialize counter
  for(i in 1:nrow(A)) # Compare the i-th row of A
  {
    if(((A$f1[i] <= x$f1) & # If A's f1 is at least as good as x's f1
        (A$f2[i] < x$f2)) || # And A's f2 is better than x's f2
       ((A$f1[i] < x$f1) & # Or if A's f1 better than x's f1
        (A$f2[i] <= x$f2))) # And A's f2 is at least as good as x's f2
      counter <- counter + 1 # Then x is dominated by the entry in A
  }
  return(counter) # Return the times the amount of times x was dominated
}
```

Thereafter I considered implementing the cooling schedule. The temperature is required to remain positive for all iterations and should never fall below zero. There are various methods of updating the temperature for simulated annealing and variant algorithms, and I only considered linear, geometric, logarithmic and exponential schemes. I gained my inspiration for these temperature update equations from the course textbook. The variables alpha and beta were chosen as specified below, based upon similar applications. Alpha is required to fall within $[0, 1]$ and beta is required to be a postive constant value. Since the linear scheme is the only scheme that has the potential to drive the temperature out of a positive range, I included a statement to cap the temperature at a minimum value as seen below The function takes as input the current temperature *temp*, the option specifying whether this is a **cool** or **reheat** problem, the type specifying the type of cooling scheme to apply [**linear, geometric, logarithmic, and exponential**], the current iteration **t**, as well as a copy of the initial temperature *temp_init*. According to the textbook by Talbi, the current iteration and initial temperature is used within the linear and logarithmic cooling schedules.

```
temp_adjust <- function(temp, option, type, t, temp_init)
{
  alpha <- 0.9 # Alpha [0,1]
  beta <- 5 # Beta > 0
  if(type == "linear")
  {
    if(option == "cool")  temp <- temp_init - t*beta # Reduces temp
    if(option == "reheat") temp <- temp_init + t*beta # Increases temp
    if(temp < 0) temp = 0.01
  }
  if(type == "geometric")
  {
    if(option == "cool")  temp <- temp*alpha # Reduces temp
    if(option == "reheat") temp <- temp*beta # Increases temp
  }
  if(type == "logarithmic")
  {
    if(option == "cool")  temp <- temp_init/log(t) # Reduces temp
    if(option == "reheat") temp <- temp_init*log(t) # Increases temp
  }
  if(type == "exponential")
  {
    if(option == "cool")  temp <- temp/(1 + beta*temp) # Reduces temp
    if(option == "reheat") temp <- temp*(1 + beta*temp) # Increases temp
  }
  return(temp)            } # Return the update temperature
```

Thereafter, I implemented a function that discards all dominated solutions from the archive A. I started the function by ensuring no duplicated rows are kept, ensuring all rows or solutions are unique. Thereafter I initialized a counter to keep track of the amount of solutions that are dominated. Then, the function uses two for loops that operate on the amount of solutions in the archive to compare the dominance of the solutions kept within the archive The first for loop represents a solution that is kept stationary while being compared to the solutions from the second for loop. Inside these two for loops, an if statement exists that checks for the dominance of the two solutions under consideration (note an measure is in place to ensure a solution is not compared to itself), by the same sort of logic that was applied previously. If this if statement returns true and is entered, the solution j must be dominated and it is added to the counter of rows/solutions that are dominated. After the nested for loop completes, the dominated rows are removed from the archive, and the archive is returned. If no dominated rows were found the input archive is returned unchanged. The function is displayed below.

```
discard_dom_solns <- function(A) # Input only archive of ND solns
{
  A <- A[!duplicated(A),] # Ensure no row/soln is duplicated
  dominated_rows <- c() # Initialize counter for dominated solutions
  for(i in 1:nrow(A)) # Compare the i-th row
  {
    for(j in 1:nrow(A)) # To the j-th row
    {
      if(i != j && # If the i-th row is not equal to the j-th row and
         ((A$f1[i] <= A$f1[j]) & # 1. If f1[i] at least as good as f1[j]
          (A$f2[i] < A$f2[j])) || # And f2[i] is better than f2[j]
         ((A$f1[i] < A$f1[j]) &  # 2. Or if f1[i] is better than f1[j]
          (A$f2[i] <= A$f2[j]))) # And f2[i] at least as good as f2[j]
        dominated_rows <- rbind(dominated_rows, j) # Then the j-th row is dominated by i-th row
    }
  }
  ifelse(is.null(dominated_rows),
         return(A),
         return(A[-sort(unique(dominated_rows)),]))
  # Output non-dominated archive
}
```

Thereafter I implemented the function to calculate the energy of the movement transition. I started by setting up the archive called A_tilde which is defined as the union of the current archive, the current solution $f(x)$, and that of the neighbouring solution $f(x')$, or $A_t = A \cup f(x) \cup f(x')$. Thereafter, I calculated the amount of dominated solutions that exist within A_tilde that dominate the solutions $f(x)$ and $f(x')$, respectively. This was accomplished by using the num_dom_soln function previously defined. Note that the input variables are received by the function in their decision form and prior to calculating the energy and the necessary pre-calculations the current and neighbouring decision values are first converted to their respective objective forms by using the evaluate_objective previously defined. This function also has a option for returning the cardinality of A_x_tilde in Step 8 of the pseudocode for DBMOSA.

```
calculate_energy <- function(A, x, x_neigh, option = "energy")
{
  A_tilde <- rbind(A, evaluate_objective(x), evaluate_objective(x_neigh)) # Set up A_tilde
  A_x_tilde <- num_dom_soln(A_tilde, evaluate_objective(x))  # Total dominating x
  A_x_tilde_neigh <- num_dom_soln(A_tilde, evaluate_objective(x_neigh)) # Total dominating x'
  A_entries <- nrow(A_tilde) # Total entries within A tilde
  energy <- (A_x_tilde_neigh - A_x_tilde)/A_entries  # Energy calculation
  if(option == "energy") return(energy) # If the energy is requested
```

```
  if(option == "A_x_tilde") return(A_x_tilde) # If the cardinality of A_x_tilde is requested
}
```

All the functions displayed above are instrumental to the working of the DBMOSA algorithm and will be called from within the DBMOSA algorithm later. The functions following are some more simple functions which will be used to illustrate results obtained from the DBMOSA algorithm. The first function is called plot_objective, and receives the archive A as input from the algorithm and plots the true pareto front in the objective space as a curved line. A black square is placed at the best possible trade-off for a global minimum or pareto optimal set of $[f_1(x), f_2(x)] = [1, 1]$ at $x = 1$, and red dots are used to illustrate where the non-dominated archived solutions are, which act as the approximate pareto optimal set.

```
plot_objective <- function(results) # Plot Pareto front within objective space
{
  true_pareto <- evaluate_objective(seq(-3,3)) # For x-values ranging from -3 to 3
  with(true_pareto, plot(f1, f2, type = "l", main = "Objective Space")) # True pareto front
  points(results[[1]]$f1, results[[1]]$f2, col = "red", pch = 1) # Archived solutions
  points(1, 1, pch = 19, col = "black", cex = 0.6) # True global minimum
}
```

The next function receives the archive A as input and then plots the approximate pareto optimal set and true pareto optimal set in decision space. A black square is placed at the best-possible true pareto optimal set, and red dots are placed at the approximate pareto optimal set. Note that this graph will contain both objective functions plotted on the same set of axes with, therefore two graphs will be visible in different sized dotted lines, and for each x-value there will be two objective function values corresponding to $f_1(x)$ and $f_2(x)$. The x-axis correspond to the x-values obtained, and the y-values reflect the values that the objective functions achieved. Note that this function uses a function called find_decision_val which will be defined after this function.

```
plot_decision <- function(results) # Plot Pareto optimal set within decision space
{
  x = seq(-3,3) # For x-values ranging from -3 to 3
  true_pareto <- evaluate_objective(x) # Evaluate the objectives
  with(true_pareto, plot(x, f1, main = "Decision Space", type = "l", lty = 3,
                         pch = 19, xlab = "X-values", ylab = "Objective Values"))
  with(true_pareto, points(x, f2, type = "l", lty = 2))
  points(find_decision_val(results[[1]]), results[[1]][[1]], col = "red", pch = 1)
  points(find_decision_val(results[[1]]), results[[1]][[2]], col = "red", pch = 1)
  points(1, 1, pch = 19, col = "black", cex = 0.6) # True global minimum
}
```

The way I implemented DBMOSA did not save the decision values and archived only the relevant objective function values. Since this quite a simple problem to solve [2 simultaneous quadratic equations], and because I did not know of a library that performs the relevant calculations, I solved the problem manually. I'm certain there is a more efficient way to do this, but since this was a simple problem I just implemented this function. The function just ensures that the roots of the polynomials are the same. The function receives the archive A as input, and returns the roots for x that are required for the relevant objective function values.

```
find_decision_val<- function(df) # Tests for + +, - +, --, +-
{
  x1 <- x2 <- c()
  for(i in 1:nrow(df)) # For each value within the results
  {
    x1[i] = sqrt(df$f1[i]) # x1 positive root
    x2[i] = sqrt(df$f2[i]) + 2 # x2 positive root
    if(!(x1[i] == x2[i])) # if x1 and x2 not equal
    {
      x1[i] = -sqrt(df$f1[i]) # Change x1 to negative root
    }
    if(!(x1[i] == x2[i])) # if x1 and x2 not equal
    {
      x2[i] = -sqrt(df$f2[i]) + 2 # Change x2 to negative root
    }
    if(!(x1[i] == x2[i])) # if x1 and x2 not equal
    {
      x1[i] = sqrt(df$f1[i]) # Change x1 back to positive root
    }
  }
  return(x1) # Return root
}
```

The following function was used to simply tabulate the results returned from the DBMOSA nicely. The function takes as input the output list of the DBMOSA algorithm, and rearranges the list to nicely to show the archive, the archive's respective decision values, the total epochs performed, the total iterations performed, as well as the final and initial decision value position, and the final temperature obtained before termination.

```
displayResults <- function(results)
{
  tab <- knitr::kable(data.frame(results[[1]], # Archive of ND solutions
                                 x = find_decision_val(results[[1]]), # Decision values
                                 epoch = results[[2]], # Epochs performed
                                 iter = results[[3]], # Iterations performed
                                 final_x = results[[4]], # Final decision value before termination
                                 init_x = results[[5]], # Initial decision value
                                 temp = results[[6]])) # Final temperature before termination
  return(list(tab)) # Returns/displays the table
}
```

Thereafter I implemented the DBMOSA algorithm by referring to the pseudocode provided to us for a minimisation MOP. The function takes as input i_max, c_max, d_max, the temperature, and the cooling/reheating schedule. The pseudocode is as follows.

0) Generate feasible solution x, initialise archive A, initialize epochs i, intialize acceptance c, initialize rejections d, intiailize iterations t.
1) If i = i_max output A and stop
2) If d = d_max then reheat temperature, increment epochs, reset acceptances and rejections. Check max epochs not reached before proceeding
3) If c = c_max then cool temperature, increment epochs, reset acceptances and rejections. Check max epochs not reached before proceeding.
4) Generate a neighbouring solution x'
5) If random number between [0, 1] > Metropolis acceptance rule; then reject neighbour, increment iterations and rejections and repeat from step 2.
6) If random number between [0, 1] < Metropolis acceptance rule; then Accept neighbour $x = x'$
7) And increment acceptances
8) And if the number of solutions dominating this x within A_tilde is zero: then insert it into the archive A and remove all solutions dominated by x.
9) And finally end the if loop by incrementing iterations and repeat from step 3.

I followed these steps precisely, and the algorithm is displayed on the following page. The function takes as input i_max, c_max, d_max, the temperature, and the cooling/reheating schedule and returns as output a list containing the archive, epochs, iterations, final decision value, initial decision value, and the final temperature that were obtained before termination. I included comments that refer to steps which correspond to steps 0 to 9 as displayed above in the pseudocode. As an initial starting position, I drew a random point drawn from a uniform distribution that lies within the limits of the decision space.

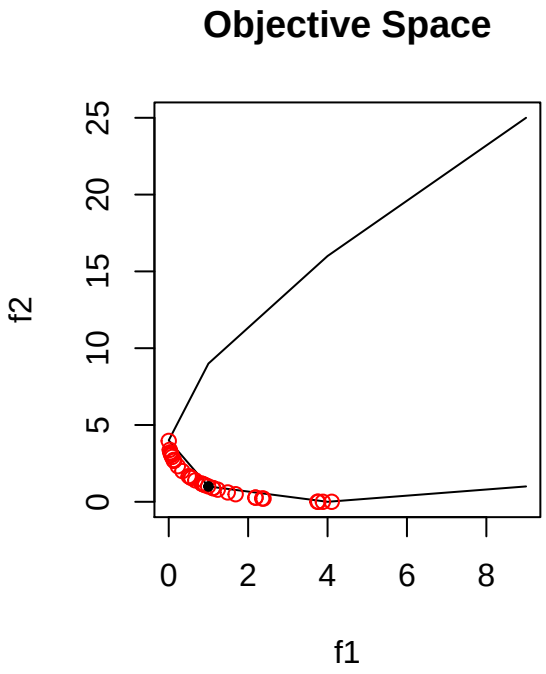
```r
DBMOSA <- function(i_max, c_max, d_max, temp, annealing_type)
{
  i <- 1 # Step 0: Initialise the number of epochs
  limits <- c(-10^5, 10^5) # Limits of the decision space
  x_init <- x <- runif(1, limits[1], limits[2]) #Step 0: Generate an initial feasible solution x
  A <- evaluate_objective(x) # Step 0: Initialise the archive A = {x}
  c <- 0 # Step 0: Initialise the number of acceptances
  d <- 0 # Step 0: Initialise the number of rejections
  t <- 1 # Step 0: Initialise the number of iterations
  temp_init <- temp # Save a copy of the initial temperature and starting position x.
  while(1) # Repeat until max iterations reached and
  {
    if(i == i_max)  return(list(A, i, t, x, x_init, temp)) # Step 1: Max iterations reached
    if(d == d_max) # Step 2: Maximum rejections reached
    {
      temp <- temp_adjust(temp, option = "heat", type = annealing_type, t, temp_init) # Step 2: Reheat
      i <- i + 1 # Step 2: Increment epochs
      c <- d <- 0 # Step 2: Reset acceptances and rejections
      if(i == i_max)  return(list(A, i, t, x, x_init, temp)) # Step 1: Max iterations reached
    }
    if(c == c_max) # Step 3: Maximum acceptances reached
    {
      temp <- temp_adjust(temp, option = "cool", type = annealing_type, t, temp_init) # Step 3: Cool
      i <- i + 1 # Step 3: Increment epochs
      c <- d <- 0 # Step 3: Reset acceptances and rejections
      if(i == i_max)  return(list(A, i, t, x, x_init, temp)) # Step 1: Max iterations reached
    }
    x_prime <- generate_neighbour(x, temp, temp_init) # Step 4: Generate Neighbour x'
    energy <- calculate_energy(A, x, x_prime) # Step 5: Calculate energy
    rand <- runif(1) # Step 5: Get random number
    if(min(1, exp(-energy/temp)) < rand) # Step 5: Random number > M.A. rule
    {
      d <- d + 1 # Step 5: Increment rejections
      t <- t + 1 # Step 5: Increment iterations
    }
    if(min(1, exp(-energy/temp)) > rand)  # Step 6: Random number < M.A. rule
    {
      x <- x_prime # Step 6: Accept Neighbour
      c <- c + 1 # Step 7: Increment acceptances
      if(calculate_energy(A, x, x_prime, option = "A_x_tilde") == 0) # Step 8: Nondominated x
      {
        A <- accept_soln(A, x_prime) # Step 8: Insert into archive A
        A <- discard_dom_solns(A) # Step 8: Remove all dominated solutions
      }
      t <- t + 1 # Step 9: Increment iterations
    } # Step 6: End of if-loop
  } # Step 9: Repeat while loop
}
```

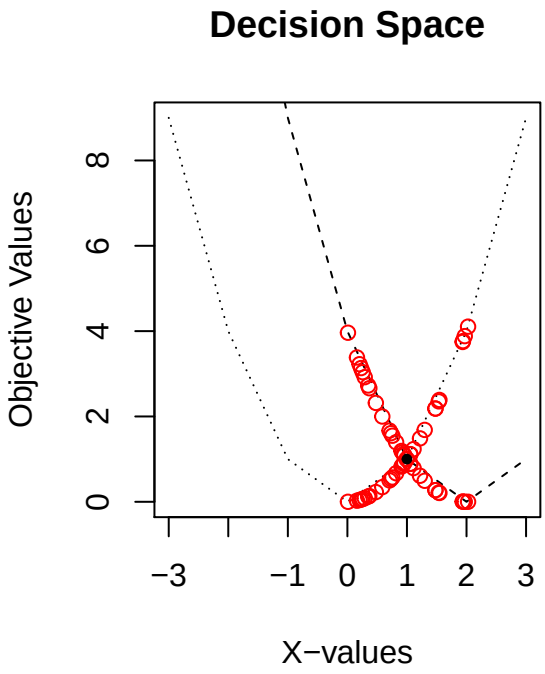
Performance investigation

After testing a few different parameters, I found that the linear and logarithmic cooling schedules performed the worst for this specific problem. The geometric and exponential parameters seemed to perform similarly and I found it hard to decide which performed better. I also found that the starting temperature T_o performed the best when I used the accept all strategy, which is concerned with setting the temperature to a high value in the start. This enables the algorithm to accept all non-improving solutions in the start, leading to a more refined exploration of the search scape. I implemented two methods to terminate the algorithm, namely a method which terminated the algorithm after a period with a very small number of acceptances, and a method which executed if the temperature reached $T_F = 0.01$. However, I found the most success with just letting the algorithm run for a pre-defined amount of iterations. This could have been due to my implementation being too strict, but I decided to continue with the pre-defined amount of epochs as the termination criterion. My algorithm's final parameters are displayed below. I gained some inspiration for these values from the article which proposed DBMOSA [Smith, Kevin & Everson, Richard & Fieldsend, Jonathan & Murphy, Chris & Misra, Ramnath. (2008). Dominance-Based Multiobjective Simulated Annealing. Evolutionary Computation, IEEE Transactions on. 12. 323 - 342. 10.1109/TEVC.2007.904345.].

```
epochs <- 200 # Similar to as found in DBMOSA article
accept <- reject <- 20 # As found in DBMOSA article
temp <- 20 # Accept all
annealing_type <- "exponential" # Cooling/reheating schedule
```

```
# Investigate exponential ------------------------------------------------
par(mfrow = c(1,2))
results <- DBMOSA(i_max = epochs,
                  c_max = accept,
                  d_max = reject,
                  temp = temp,
                  annealing_type = annealing_type)
plot_objective(results)
plot_decision(results)
```






```
print(displayResults(results))
```

```
## [[1]]
##
##
##                f1          f2          x   epoch   iter     final_x     init_x        temp
## ----   ----------  ----------  ----------  ------  -----  ----------  ---------  ----------
## 1       0.9920001   1.0080320   0.9959920     200   4495   0.9440666   23260.14   0.0124922
## 13      0.5708952   1.5485910   0.7555760     200   4495   0.9440666   23260.14   0.0124922
## 11      0.5354192   1.6085255   0.7317234     200   4495   0.9440666   23260.14   0.0124922
## 15      0.0417289   3.2246226   0.2042766     200   4495   0.9440666   23260.14   0.0124922
## 12      3.7458053   0.0041721   1.9354083     200   4495   0.9440666   23260.14   0.0124922
## 16      0.1357128   2.6621442   0.3683921     200   4495   0.9440666   23260.14   0.0124922
## 17      0.8529734   1.1587110   0.9235656     200   4495   0.9440666   23260.14   0.0124922
## 18      4.1048004   0.0006776   2.0260307     200   4495   0.9440666   23260.14   0.0124922
## 19      1.4860855   0.6098813   1.2190511     200   4495   0.9440666   23260.14   0.0124922
## 110     0.5014062   1.6690046   0.7081004     200   4495   0.9440666   23260.14   0.0124922
## 111     0.9130214   1.0909352   0.9555215     200   4495   0.9440666   23260.14   0.0124922
## 112     2.3534424   0.2170690   1.5340934     200   4495   0.9440666   23260.14   0.0124922
## 113     0.3435965   1.9989124   0.5861710     200   4495   0.9440666   23260.14   0.0124922
## 114     1.2345430   0.7901434   1.1110999     200   4495   0.9440666   23260.14   0.0124922
## 115     0.8252722   1.1914925   0.9084449     200   4495   0.9440666   23260.14   0.0124922
## 116     0.0000874   3.9627015   0.0093465     200   4495   0.9440666   23260.14   0.0124922
## 14      0.0660180   3.0382595   0.2569396     200   4495   0.9440666   23260.14   0.0124922
## 117     3.7665265   0.0035101   1.9407541     200   4495   0.9440666   23260.14   0.0124922
## 118     0.6694710   1.3966226   0.8182121     200   4495   0.9440666   23260.14   0.0124922
## 119     0.0258662   3.3825473   0.1608297     200   4495   0.9440666   23260.14   0.0124922
## 120     0.0527540   3.1340244   0.2296824     200   4495   0.9440666   23260.14   0.0124922
```

```
## 121    3.8830809   0.0008671   1.9705534    200   4495   0.9440666   23260.14   0.0124922
## 122    2.3873982   0.2069151   1.5451208    200   4495   0.9440666   23260.14   0.0124922
## 123    1.1139071   0.8922352   1.0554180    200   4495   0.9440666   23260.14   0.0124922
## 124    2.1841782   0.2725920   1.4778966    200   4495   0.9440666   23260.14   0.0124922
## 125    1.1176778   0.8888665   1.0572028    200   4495   0.9440666   23260.14   0.0124922
## 126    0.1217783   2.7259082   0.3489675    200   4495   0.9440666   23260.14   0.0124922
## 127    1.6848335   0.4927880   1.2980114    200   4495   0.9440666   23260.14   0.0124922
## 128    0.2280709   2.3178003   0.4775676    200   4495   0.9440666   23260.14   0.0124922
## 129    2.1859080   0.2719814   1.4784817    200   4495   0.9440666   23260.14   0.0124922
## 130    0.0847365   2.9203550   0.2910954    200   4495   0.9440666   23260.14   0.0124922
## 131    0.8912618   1.1149953   0.9440666    200   4495   0.9440666   23260.14   0.0124922
```

The results are displayed above, and by using the functions that were declared a table is presented with the output information. First, the two relevant objective function values are displayed as **f1** and **f2**. The corresponding decision value is presented next to these objective function values under the label **x**. The total amount of epochs and iterations that were performed are displayed under **epoch** and **iter** respectively. The final decision value that was reached is also displayed under **final_x**, and the initial decision value is displayed under **init_x** and the final temperature is also displayed under **temp**. Therefore by inspecting the results, I believe this algorithm could have performed better in terms of spread and convergence. This will be improved by investigating three diversity preservation techniques, namely nearest-neighbour, histogram and kernel methods.

Diversity Preservation Techniques

Nearest-neighbour method

The nearest-neighbour method is a method that is commonly employed to limit the loss of diversity in the neighbourhood functions of P-metaheuristics. This method deteriorates solutions that have high densities within the neighbourhood functions by using the crowding distance metric and the non-dominance ranking of the solutions. From what I gathered from the textbook by Talbi, the following flow applies to the nearest neighbour method, and I used it as a sort of pseudocode. Note this is called at each iteration of the algorithm when the neighbourhood function is called.

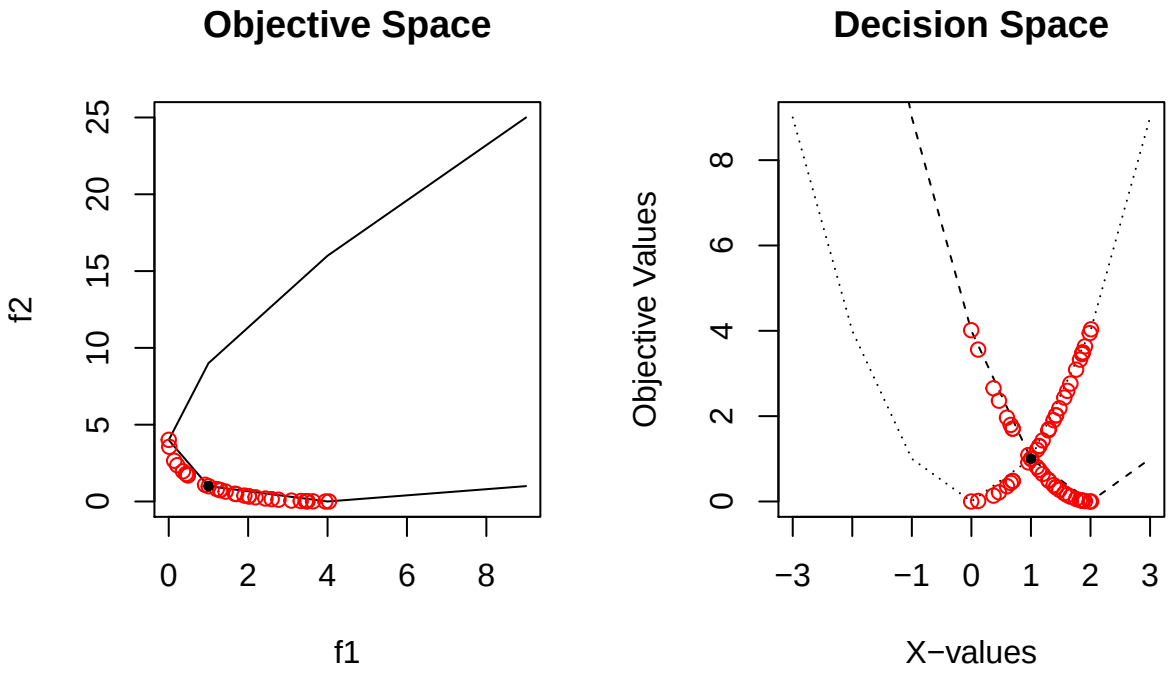
1) Generate population with population size p
2) Rank population according to non-dominance ranking
3) Calculate crowding distance for solutions [circumference of rectangle containing neighbours]
4) Perform a tournament selection, picking solutions randomly from population.
5) Choose the highest non-dominated ranking.
6) If more than one non-dominated solution exists within the best non-dominated rank, choose the neighbour with the highest crowding distance [this is the part that ensures diversity]

The DBMOSA algorithm flow remained the same; the neighbourhood function is all that was adapted. The code is displayed below. I used a population size of 7 and a tournament size of 2. The population was generated with perturbations around the current solution, after which the crowding distance was calculated for each entry of the population and the original solution x by using the crowding_distance function from the emoa library. Thereafter, the objective function values were sorted by a function fastNonDominatedSorting available from the nsga2R library, which ranks non-dominated solutions for minimisation MOPs and returns a list containing the indices and their respective rankings. I then assigned these ranking to each entry of the dataframe from within a for loop, and then proceeded to perform touranament selection. A selection size of k entries was sampled without replacement and the corresponding entries were taken from the population to create the tourney variable. I then chose the variable from this tourney as the variable which exhibited the highest ranking. Since there may be more than one entry which has the maximum ranking, I then further specified that the variable chosen should have the maximum crowding distance that was calculated earlier. The combination of these two selections ensure that only one neighbour is returned, and the function then returns the generated neighbouring value.

```
# NNM ---------------------------------------------------------------------
generate_neighbour <- function(x, temp, temp_init)
{
  p <- 7 # Population size p
  limits <- c(-10^2, 10^2) # Limits of perturbation
  k <- 2 # Tournament selection size k
  population <- x + runif(p, limits[1], limits[2]) # Generate new population
  x_values = c(x, population) # Total population including original x
  distances <- emoa::crowding_distance(matrix(x_values, ncol = p + 1)) # Crowding distances
  population <- data.frame(x_values, distances) # Merge distances with x
  population$ranking <- Inf # Initiaize ranking variable
  ranking <- nsga2R::fastNonDominatedSorting(evaluate_objective(population$x_values)) # Returns ranking
  for(i in 1:length(ranking)) # Assigns the rank to indices
  {
    population[ranking[[i]],]$ranking <- i # Assigns the rank to indices
  }
  tourney <- population[sample.int(nrow(population), size = k),] # Sample k random population
  tourney <- tourney[tourney$ranking == min(tourney$ranking),] # Select lowest ND ranking
  tourney <- tourney[tourney$distances == max(tourney$distances),] # Select highest distance
```

```
  x_new <- tourney[1,1] # Access x_value
  return(x_new) # Return x_value
}
```

```
# Investigate NN ------------------------------------------------------------
par(mfrow = c(1,2))
results <- DBMOSA(i_max = epochs,
                  c_max = accept,
                  d_max = reject,
                  temp = temp,
                  annealing_type = annealing_type)
plot_objective(results)
plot_decision(results)
```




```
print(displayResults(results))
```

```
## [[1]]
##
##
##                f1          f2           x   epoch   iter    final_x     init_x        temp
## ----   ----------  ----------  ----------  ------  -----  ---------  ---------  ----------
## 12      0.3572424   1.9664522   0.5976975     200   5109   1.380647   94340.21   0.0019229
## 11      1.2945307   0.7434329   1.1377744     200   5109   1.380647   94340.21   0.0019229
## 13      3.0898269   0.0586655   1.7577903     200   5109   1.380647   94340.21   0.0019229
## 15      1.4326437   0.6449199   1.1969309     200   5109   1.380647   94340.21   0.0019229
## 1       0.0128117   3.5600569   0.1131887     200   5109   1.380647   94340.21   0.0019229
## 17      0.9154486   1.0882854   0.9567908     200   5109   1.380647   94340.21   0.0019229
```

```
## 18     1.0082284   0.9918053    1.0041058     200   5109   1.380647   94340.21   0.0019229
## 19     0.4813276   1.7062164    0.6937778     200   5109   1.380647   94340.21   0.0019229
## 110    0.1376953   2.6534024    0.3710732     200   5109   1.380647   94340.21   0.0019229
## 16     3.4955918   0.0169911    1.8696502     200   5109   1.380647   94340.21   0.0019229
## 14     4.0359090   0.0000802    2.0089572     200   5109   1.380647   94340.21   0.0019229
## 112    2.4348802   0.1932391    1.5604103     200   5109   1.380647   94340.21   0.0019229
## 113    3.4544144   0.0199924    1.8586055     200   5109   1.380647   94340.21   0.0019229
## 114    0.2145268   2.3618453    0.4631704     200   5109   1.380647   94340.21   0.0019229
## 115    1.2107807   0.8093614    1.1003548     200   5109   1.380647   94340.21   0.0019229
## 116    1.6669319   0.5025432    1.2910972     200   5109   1.380647   94340.21   0.0019229
## 117    2.5929299   0.1518991    1.6102577     200   5109   1.380647   94340.21   0.0019229
## 118    3.4845768   0.0177683    1.8667021     200   5109   1.380647   94340.21   0.0019229
## 119    2.0196475   0.3350754    1.4211430     200   5109   1.380647   94340.21   0.0019229
## 120    3.9452821   0.0001884    1.9862734     200   5109   1.380647   94340.21   0.0019229
## 121    0.4357947   1.7952048    0.6601475     200   5109   1.380647   94340.21   0.0019229
## 122    3.6349213   0.0087335    1.9065470     200   5109   1.380647   94340.21   0.0019229
## 123    1.6991294   0.4851031    1.3035066     200   5109   1.380647   94340.21   0.0019229
## 124    2.1852403   0.2722170    1.4782558     200   5109   1.380647   94340.21   0.0019229
## 125    0.0000133   4.0146055   -0.0036480     200   5109   1.380647   94340.21   0.0019229
## 111    3.3248917   0.0311775    1.8234286     200   5109   1.380647   94340.21   0.0019229
## 126    2.7642787   0.1138307    1.6626120     200   5109   1.380647   94340.21   0.0019229
## 127    0.4811290   1.7065905    0.6936346     200   5109   1.380647   94340.21   0.0019229
## 128    1.3008356   0.7386684    1.1405418     200   5109   1.380647   94340.21   0.0019229
## 129    2.0176351   0.3358958    1.4204348     200   5109   1.380647   94340.21   0.0019229
## 130    1.9061862   0.3835981    1.3806470     200   5109   1.380647   94340.21   0.0019229
```

Histogram method

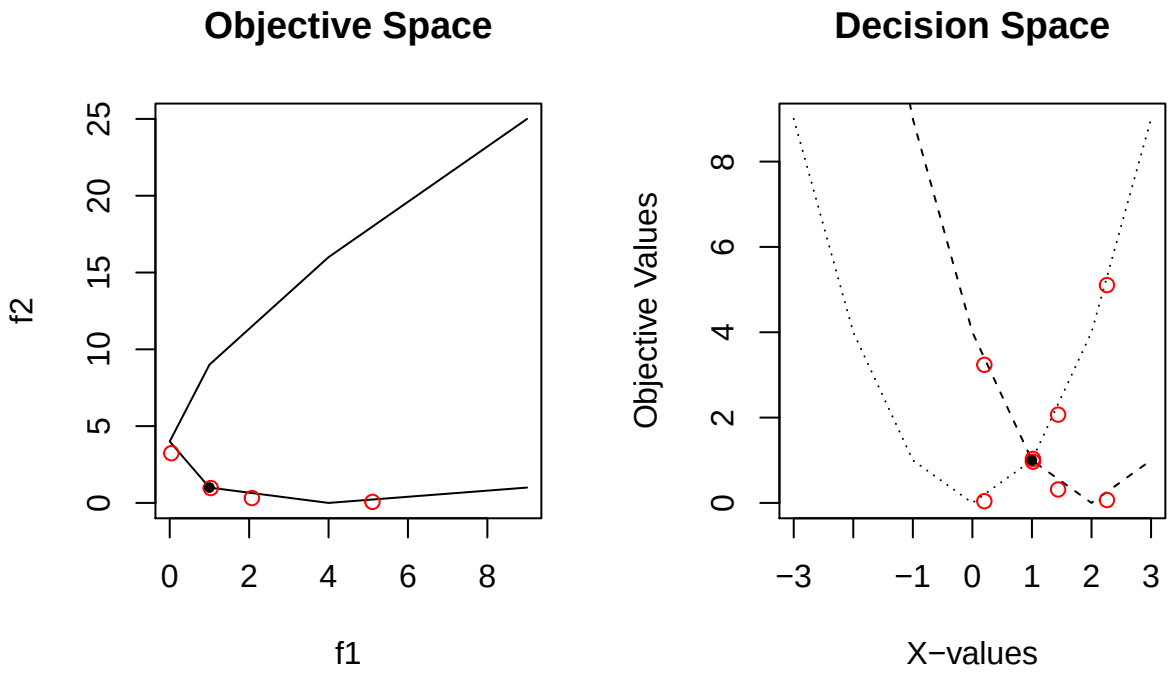
The histogram method is another method that is employed to limit the loss of diversity when generating new neighbours. This method deteriorates solutions that have high densities, meaning they are bundled together; by applying partitions within the objective space. The neighbour will be selected as the best solution found within the most sparse partition of the search space. Note that there are applications which apply these histogram-like partitions to the decision space, but most applications seem to lean towards partitioning the objective space. Since this problem only contains one decision variable, I also partitioned the objective space as it seems more viable. By employing these partitions, the neighbourhood function allows for the encouragement of exploring diverse solutions. From what I gathered in the textbook by Talbi, the following four steps were set up.

1) Generate population.
2) Split population into partitions.
3) Find most sparse partition that still contains entries.
4) Output the neighbour as the best solution within this sparse partition.

I implemented a partition size of 4, effectively splitting the objective space into 4 different areas as separated by the horizontal_split and vertical_split variables. I also implemented a population size of 50, with perturbations being limited within [-1000, 1000]. After splitting the quadrants, I ensured no quadrants are empty, and after removing any potential invalid quadrants I chose the smallest or most sparse quadrant, as denoted by least_dense_quad, to be the the partition that would provide the most diversification. In some cases, least_dense_quad only contained one entry, if which it returned the decision value of the corresponding point. However, if the point did contain more than one point, I chose the point with the highest non-dominance ranking by implementing the same function as before, fastNonDominatedSorting from the nsga2R library.

```
# Histograms ------------------------------------------------------------
generate_neighbour <- function(x, temp, temp_init)
{
  p <- 7 # Population size p
  limits <- c(-10^3, 10^3) # Limits of perturbation
  x <- x + runif(p, limits[1], limits[2]) # Generate new population
  population <- evaluate_objective(x) # Find objective values
  horizontal_split <- mean(population$f2) # Find horizontal split point
  vertical_split <- mean(population$f1) # Find vertical split point
  # Quadrants
  upper_right <- subset(population, f1 > vertical_split & f2 > horizontal_split)
  lower_right <- subset(population, f1 > vertical_split & f2 < horizontal_split)
  upper_left <- subset(population, f1 < vertical_split & f2 > horizontal_split)
  lower_left <- subset(population, f1 < vertical_split & f2 < horizontal_split)
  quadrants <- list(upper_right, lower_right, upper_left, lower_left) # Combine quadrants
  sizes <- sapply(quadrants, nrow) # Get size of all quadrants
  valid_idx <- which(sizes > 0) # Get all non-empty quadrants
  quadrants <- quadrants[valid_idx] # Subset all non-empty quadrants
  sizes <- sapply(quadrants, nrow) # Get size of all **valid** quadrants
  least_dense_quad <- data.frame(quadrants[which.min(sizes)]) # Find most sparse quadrant
  if(nrow(least_dense_quad) == 1) return(find_decision_val(least_dense_quad)) # Return decision value
  ranking <- nsga2R::fastNonDominatedSorting(least_dense_quad) # If more than one entry
  return(find_decision_val(least_dense_quad[ranking[[1]],])) # Return the non-domim decision value
}
```

```r
# Investigate Hist ----------------------------------------------------------
par(mfrow = c(1,2))
results <- DBMOSA(i_max = epochs,
                  c_max = accept,
                  d_max = reject,
                  temp = temp,
                  annealing_type = annealing_type)
plot_objective(results)
plot_decision(results)
```




```r
print(displayResults(results))
```

```
## [[1]]
##
##
##              f1          f2          x   epoch   iter    final_x      init_x        temp
## ---  ----------  ----------  ----------  ------  -----  ---------  ----------  ----------
## 12    1.0329862   0.9675490   1.0163593     200   4047   1.438435   -54648.54   0.0664452
## 13    0.0403149   3.2371721   0.2007857     200   4047   1.438435   -54648.54   0.0664452
## 1     5.1055185   0.0673607   2.2595394     200   4047   1.438435   -54648.54   0.0664452
## 11    2.0690944   0.3153556   1.4384347     200   4047   1.438435   -54648.54   0.0664452
```

Kernel method

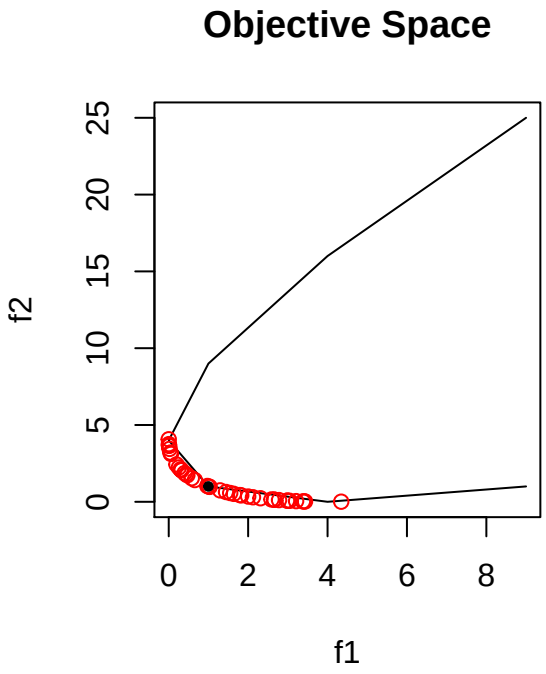
The kernel method is a method is employed to limit the loss of diversity when generating new neighbours. This method generates a population around solution i, and uses a kernel function to get an estimate on the density of solutions. Several variants of this method exist, and I employed the standard kernel method which uses the sum of the distance between the points as the kernel function to provide an estimation of the density of the solutions. Thereafter, the solution with the lowest density can be selected as the neighbour to output. From what I could gather from the textbook, the method works as follows.

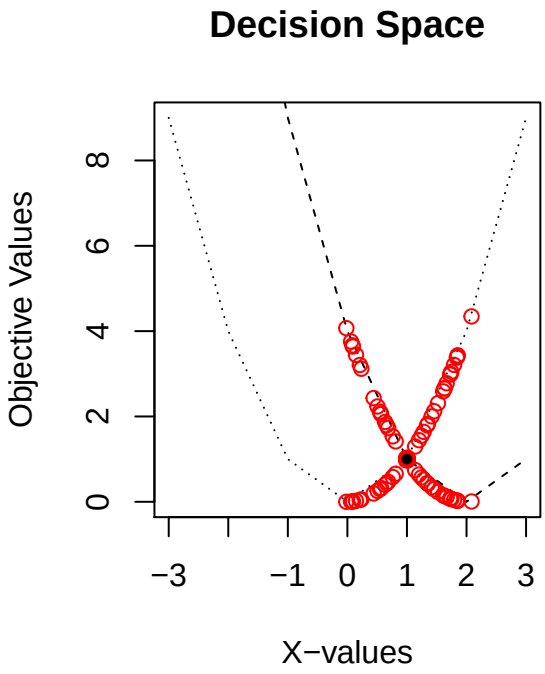
1) Generate population of perturbed points
2) Find density estimate using distance between points
3) Return the least dense solution

I started the method by declaring a population size of 200, which are perturbations of the current solution. Thereafter, I evaluated the objective function and calculated the distances between the solutions to get an estimate on the density. I then chose the lowest density, which will aid to diversify the objective space and the function then outputs the corresponding decision value.

```
# Kernel methods ------------------------------------------------------------
generate_neighbour <- function(x, temp, temp_init)
{
  p <- 200 # Population size p
  limits <- c(-10^3, 10^3) # Limits of perturbation
  x <- x + runif(p, limits[1], limits[2]) # Perturb points
  population <- evaluate_objective(x) # Generate population
  distance_matrix <- as.matrix(dist(x, diag = T, upper = T)) # Distance between solns
  density_estimate <- rowSums(distance_matrix) # Density estimate of all solutions
  x_new <- find_decision_val(population[which.min(density_estimate),]) # Choose lowest density
  return(x_new) # Output neighbour
}
```

```
# Investigate kernel --------------------------------------------------------
par(mfrow = c(1,2))
results <- DBMOSA(i_max = epochs,
                  c_max = accept,
                  d_max = reject,
                  temp = temp,
                  annealing_type = annealing_type)
plot_objective(results)
plot_decision(results)
```






```
print(displayResults(results))
```

```
## [[1]]
##
##
##                f1          f2            x   epoch   iter    final_x      init_x         temp
## ----   ----------  ----------  -----------  ------  -----  ---------  ----------  ----------
## 2       0.3998122   1.8705841    0.6323070     200   4517   1.726056   -33608.82   0.0117578
## 11      3.4092604   0.0235874    1.8464183     200   4517   1.726056   -33608.82   0.0117578
## 1       0.3207658   2.0553181    0.5663619     200   4517   1.726056   -33608.82   0.0117578
## 12      0.2554573   2.2337460    0.5054278     200   4517   1.726056   -33608.82   0.0117578
## 13      3.3799801   0.0260912    1.8384722     200   4517   1.726056   -33608.82   0.0117578
## 15      0.4368713   1.7930216    0.6609624     200   4517   1.726056   -33608.82   0.0117578
## 16      0.5798037   1.5340101    0.7614484     200   4517   1.726056   -33608.82   0.0117578
## 17      1.5478654   0.5713358    1.2441324     200   4517   1.726056   -33608.82   0.0117578
## 18      1.7915637   0.4375914    1.3384931     200   4517   1.726056   -33608.82   0.0117578
## 19      2.6557019   0.1371721    1.6296324     200   4517   1.726056   -33608.82   0.0117578
## 110     0.3160210   2.0673911    0.5621575     200   4517   1.726056   -33608.82   0.0117578
## 111     0.0548323   3.1181807    0.2341629     200   4517   1.726056   -33608.82   0.0117578
## 14      3.0471365   0.0647169    1.7456049     200   4517   1.726056   -33608.82   0.0117578
## 112     0.0038889   3.7544453    0.0623609     200   4517   1.726056   -33608.82   0.0117578
## 113     4.3444576   0.0071126    2.0843362     200   4517   1.726056   -33608.82   0.0117578
## 114     0.0003021   4.0698272   -0.0173813     200   4517   1.726056   -33608.82   0.0117578
## 115     0.9715782   1.0288315    0.9856867     200   4517   1.726056   -33608.82   0.0117578
## 116     3.4330495   0.0216534    1.8528490     200   4517   1.726056   -33608.82   0.0117578
## 117     2.6594094   0.1363311    1.6307696     200   4517   1.726056   -33608.82   0.0117578
## 118     0.4690047   1.7296479    0.6848392     200   4517   1.726056   -33608.82   0.0117578
## 119     2.5878588   0.1531295    1.6086823     200   4517   1.726056   -33608.82   0.0117578
```

```
## 120    3.2078791   0.0436579   1.7910553    200   4517   1.726056   -33608.82   0.0117578
## 121    1.8264564   0.4205982   1.3514645    200   4517   1.726056   -33608.82   0.0117578
## 122    1.6394408   0.5178148   1.2804065    200   4517   1.726056   -33608.82   0.0117578
## 123    0.0072521   3.6666153   0.0851592    200   4517   1.726056   -33608.82   0.0117578
## 124    2.1215823   0.2953214   1.4565652    200   4517   1.726056   -33608.82   0.0117578
## 125    0.1942157   2.4314195   0.4406990    200   4517   1.726056   -33608.82   0.0117578
## 126    0.2986226   2.1127676   0.5464638    200   4517   1.726056   -33608.82   0.0117578
## 127    0.0208107   3.4437744   0.1442591    200   4517   1.726056   -33608.82   0.0117578
## 128    0.0085632   3.6384128   0.0925376    200   4517   1.726056   -33608.82   0.0117578
## 129    0.0439426   3.2054431   0.2096249    200   4517   1.726056   -33608.82   0.0117578
## 130    2.0016619   0.3424579   1.4148010    200   4517   1.726056   -33608.82   0.0117578
## 131    1.4435523   0.6376354   1.2014792    200   4517   1.726056   -33608.82   0.0117578
## 132    2.7783686   0.1109930   1.6668439    200   4517   1.726056   -33608.82   0.0117578
## 133    1.0305910   0.9698699   1.0151803    200   4517   1.726056   -33608.82   0.0117578
## 134    0.6568578   1.4149872   0.8104676    200   4517   1.726056   -33608.82   0.0117578
## 135    1.0151362   0.9849775   1.0075397    200   4517   1.726056   -33608.82   0.0117578
## 136    1.2944586   0.7434875   1.1377428    200   4517   1.726056   -33608.82   0.0117578
## 137    2.9955547   0.0724864   1.7307671    200   4517   1.726056   -33608.82   0.0117578
## 138    2.3108726   0.2302508   1.5201555    200   4517   1.726056   -33608.82   0.0117578
## 139    2.9792709   0.0750451   1.7260564    200   4517   1.726056   -33608.82   0.0117578
```

Conclusion

The original DBMOSA algorithm could have performed better in my opinion, but after the inclusion of the diversity-based methods the performance was improven significantly. The nearest neighbour and kernel-based diversity preservation techniques can be seen to have a better spread and convergence than the original DBMOSA algorithm, and the histogram method seems to be worse than the other two methods. However, I confess that depending on the run of execution the histogram method might have even performed worse than the original DBMOSA algorithm. I reconciled myself by accepting that this is either due to my implementation of the horizontal and vertical quadrant-splitting planes, or it is due to the shape of the objective space being non-optimal for splitting into four quadrants with respect to the minimum and maximum objective values of the axes. I do believe that the nearest neighbour method performed the best as it had the best spread with the best convergence on the pareto front than all the other diversity-based methods and the original DBMOSA algorithm. The kernel method is a close second, and the third best performer is either the histogram method or the original DBMOSA method, depeneding on the seed of the execution.