Optimisation 874

Post Block Assessment 3

Francois van Zyl: 18620426

14 August 2020

Hybrid PSO

Introduction

The goal of this report is to devise a hybrid PSO meta-heuristic which contains a high- and low-level PSO algorithm. The high-level PSO algorithm will be responsible for finding the optimal parameters ω, C_1, C_2 for the low-level PSO algorithm to navigate the search space, and the low-level PSO algorithm will be responsible for solving an optimization problem to optimality or at least near-optimality. The decision space is constrained to $-512 \leq x, y \leq 512$ and the global optimum exists at $f(512, 404.2319) = -959.6407$. The assignment detail specifies that the high-level PSO must employ 10 particles with constant search parameters $\omega = 1$, $C_1 = 1$, and $C_2 = 1$. The low-level PSO must employ 50 particles to solve the underlying decision space, and it must use the search parameters provided to it from the high-level PSO. Therefore, it is important to note that the search space of the high-level PSO pertains to the parameters of the low-level PSO, and the search space of the low-level PSO pertains to the decision space of x and y. Both PSO's will employ a dynamic stopping criterion, which will terminate the individual algorithms after a pre-defined amount of iterations has passed without observing any improvement in the global best solution. The objective function for this problem is defined by the following equation, $f(x, y) = -(y + 47)sin(\sqrt{|y + x/2 + 47|}) - xsin(\sqrt{|x - (y + 47)|})$. Since this is a somewhat convoluted problem, with two algorithms operating within another, I kept the program as modular as I could by implementing 6 functions for the first question of the assignment, and 3 functions for the second question of the assignment. A list is provided at a later stage in the report that will summarize the functions, but I will start my discussion of this assignment with an in detail explanation of the code I implemented.

- Note: The low-level swarm is consistently assigned to the object named particles, and the high-level swarm is consistently assigned to the object named parameters.

Functions

The first function that I implemented is a function that pertains to evaluating the previously defined objective function that the low-level PSO is responsible for solving. The function takes as input either scalars or vectors x and y, and returns the calculated objective function value.

```
# Evaluate objective definition ---------------------------------------------
rm(list = ls()) # Clear workspace
evaluate_objective <- function(x, y) # Input x and y
{
  f <- (-1)*(y + 47) * sin(sqrt(abs(y + x/2 + 47))) - x * sin(sqrt(abs(x - (y + 47))))
  return(f) # Return calculated objective function value/values
}
```

The following function is not essential to the implementation of the hybrid meta-heuristic, but it will aid in investigating the search space and performance of the low-level PSO. The function serves two purposes, denoted by *options*, where the first option, *space* relates to plotting the entire decision space to get an understanding of the search space. To set this 3D plot up, I used some work from an article found at jamesmccaffrey.wordpress.com, which is hyper-linked if the reader would wish to inspect it. The first option declares the search space for x and y, then evaluates the respective objective function values at each possible combination of x and y. Note that both x and y are evaluated as a sequence of a certain length, and therefore the corresponding combination matrix z is a square matrix with the same dimensions as the sequence length of x and y. A color grid is then set up as a ramp palette with the specified colors, and the colors are cut into ranges according to the specified palette where the variable *facetcol* contains the value of the colors to use for the respective z-value obtained. The second option, *swarm* serves a similar function to the first option; the 3D plot of the decision space is drawn in gray, but now the final points of the low-level PSO are added to the graph in red squares with crosses in them. The final particles of the low-level PSO are returned from the function evaluate_PSO, which will be considered later, but for now we can just notice that the evaluate_PSO function takes as input the optimal or near-optimal parameters found by the high-level PSO; in conjunction with an option variable, the variable k_{max}, which specifies the termination criterion – a maximum amount of iterations that can pass without seeing an improvement in the global best found solution p_g.

```r
# Check decision space  ---------------------------------------------------
plot_search <- function(phi, theta, parameters, option)
{ # Input: Viewing angles, parameters to test low-level PSO, and option of plotting
  # Output 1: Graph showing decision space
  # Output 2: Graph showing decision space with low-level PSO final particles
  if(option == "space") # Plot decision space with the global optimum easily viewable
  {
    x <- y <-  seq(-512,512, length = 100) # Search space
    z <- outer(x, y, evaluate_objective) # Evaluate search space combinations
    nrz <- nrow(z) #  Number of rows, 100
    ncz <- ncol(z) # Number of columns, 100
    jet.colors <- colorRampPalette(c("midnightblue","blue",
                                     "cyan","green", "yellow", # Source: Hyperlink
                                     "orange", "red", "darkred")) # Set up color grid
    nbcol <- 32 # Number of colors to use
    color <- jet.colors(nbcol) # Set up colors
    zfacet <- z[-1,-1] + z[-1,-ncz] + z[-nrz,-1] + z[-nrz,-ncz] # z-values for sections
    facetcol <- cut(zfacet, nbcol) # Cut z-values according to colors
    persp(x,y,z, col=color[facetcol], phi=phi, theta=theta, # Plot the 3D graph
          ticktype="detailed", d = 5, r = 1, expand = 0.4) # Source: Hyperlink
  }
  if(option == "swarm") # Plot decision space from the front and bottom
  {
    particles <- evaluate_PSO(parameters, k_max, option = "low_level") # Evaluate low-level PSO
    x <- y <-  seq(-512,512, length = 100) # Search space
    z <- outer(x, y, evaluate_objective) # Evaluate Objective Combinations
    pmat <- persp(x,y,z, col="gray", phi=phi, theta=theta, # Plot the 3D graph in gray
                  ticktype="detailed", d = 5, r = 1, expand=0.4) # Source: Hyperlink
    new_points <- trans3d(x = particles$x, y = particles$y, z = particles$f, pmat) # Project particles
    points(new_points, pch = 7, col = "red", cex = 1.6) # Add particles to 3D plot
    particles$v_x <- particles$v_y <- NULL # Remove velocity for viewing ease
    return(particles) # Returns the particles for inspection of the results
  }
}
```

PSO algorithms usually initialize the entire swarm's locations and velocities randomly, and from these initialized locations, the locally best found solutions p_i and globally best found solution p_g are initialized. The function I implemented to perform this initialization is displayed below, and it takes as input two variables, namely the number of particles that are to be initialized, and the option that specifies whether the current initialization is for the high- or low-level PSO algorithm.

```
# Randomly initialize particles ---------------------------------------------
init_PSO <- function(num_particles, option)
{ # Input: Number of particles within swarm: (10) or (50)
  # Input: Option specifying whether this is the high- or low-level PSO
  # Output: Initialized particles for low-level or high-level PSO
  if(option == "low_level") # Low-level PSO initialization
  {
    x_lim <- y_lim <- c(-512, 512) # Boundaries of search space
    v_x <- v_y <- x <- y <- p_i_x <- p_i_y <- f <-  c() # Initialize variables
    for(i in 1:num_particles) # For each particle (50)
    {
      p_i_x[i] <- x[i] <- runif(1, min = x_lim[1], max = x_lim[2]) # Initialize x-position
      p_i_y[i] <- y[i] <- runif(1, min = y_lim[1], max = y_lim[2]) # Initialize y-position
      v_x[i] <- x[i]*init_low # Initialize x-velocity as one-tenth of x-position
      v_y[i] <- y[i]*init_low # Initialize y-velocity as one-tenth of y-position
    }
    f <- evaluate_objective(x, y) # Evaluate the objectives of the current position
    best_x <- x[which.min(f)] # Store the x-value with the lowest objective value
    best_y <- y[which.min(f)] # Store the y-value with the lowest objective value
    best_f <- min(f) # Store the lowest objective value
    particles <- data.frame(x, y, v_x, v_y, p_i_x, p_i_y, f, best_x, best_y, best_f)
    return(particles) # Returns dataframe of initialized particles
  }
  if(option == "high_level") # High-level PSO initialization
  {
    # Initialize variables
    p_i_omega <-  p_i_c_1 <- p_i_c_2 <-  v_omega <- omega <-  v_c_1 <- c_1 <- c_2 <- v_c_2 <- c()
    for(i in 1:num_particles)  # For each particle (10)
    {
      p_i_omega[i] <- omega[i] <- round(runif(1, min = 0.4, max = 0.9), digits = 1) # Initialize omega
      p_i_c_1[i] <- c_1[i] <- round(runif(1, min = 0.5, max = 2), digits = 1) # Initialize c_1
      p_i_c_2[i] <- c_2[i] <- round(runif(1, min = 0.5, max = 2), digits = 1) ## Initialize c_2
      v_omega[i] <- omega[i] * init_high # Initializes omega velocity
      v_c_1[i]<- c_1[i] * init_high # Initializes c_1 velocity
      v_c_2[i]<- c_2[i] * init_high ## Initializes c_2 velocity
    } # Now merge the initialized velocities, positions, and local best into a dataframe parameters.
    parameters <- data.frame(omega, c_1, c_2, v_omega, v_c_1, v_c_2, p_i_omega, p_i_c_1, p_i_c_2)
    parameters <- evaluate_PSO(parameters, k_max = k_max, option = "high_level") # Evaluate parameters
    best_omega <- parameters[which.min(parameters$f_avg),]$omega # Store the best parameter's omega
    best_c_1 <- parameters[which.min(parameters$f_avg),]$c_1 # Store the best parameter's c_1
    best_c_2 <- parameters[which.min(parameters$f_avg),]$c_2 # Store the best parameter's c_2
    best_f <- min(parameters$f_avg) # Store the lowest objective value
    parameters <- (cbind(parameters, best_omega, best_c_1, best_c_2, best_f)) # Merge parameters
    return(parameters) # Returns dataframe of initialized particles
  }
}
```

For the case of the low-level PSO option, the algorithm starts by defining the search space boundaries and initializing the x and y positions, velocities, and locally best found positions. Values are assigned to these variables within a for-loop where it can be noted that the initial x and y positions are drawn from a uniform distribution that is constrained to the previously mentioned search space boundaries. It should be noted that the locally best found solutions are initialized to being the same values as the initialized x and y positions, as is customary for PSO algorithms at the 0-th iteration. I consulted available literature, which is hyper-linked if the reader wishes to review it, which provided motivation for me to initialize the velocity as a multiple of the current position and a certain velocity initialization factor. For zero-centered search spaces, this initialization factor usually assumes values within the range of $0.1 \leq k_{init} \leq 1.0$, which will ensure that the majority of the result of the position and velocity are initialized in such a way that they do not move out of the search space. However, particles will still be able to move out of the search space, since the magnitude of the velocity is not the only contributing factor to leaving the pre-defined search space. This concern will be addressed in the update_PSO function. After initializing all the positions, locally best found positions, and velocities; all the entries objective function values were calculated. The entry with the lowest objective function value's position and objective function value is saved as the globally best found solution. The function then binds the position, velocity, local best solutions, objective functions and global best solutions into a dataframe and returns it as the initialization of the particles for the low-level PSO.

For the case of the high-level PSO option, it is important to note that the algorithm is no longer centrally defined by 50 particles' individual solutions with a search space defined by x and y. Instead it is defined by 10 particles where the individual solutions represent the combination of parameters ω, C_1, and C_2. For the low-level PSO, the performance is defined by evaluating the individual particle positions at the pre-defined objective function. Since the high-level PSO needs to be able to assess the performance of the underlying PSO algorithm, and I decided to assess the performance of the combinations of parameters by taking the average of the 50 objective function values returned from the 10 different low-level PSO implementations. To perform the algorithm execution for the low-level PSO, I implemented a function evaluate_PSO which will be covered later, but for now it is only necessary to note that this function takes as input the combination of parameters, the maximum iterations to reiterate without improvement in the globally best found solution, as well as an option to specify whether this is evaluation is for the high- or low-level PSO algorithm. With the high-level option specified, evaluate_PSO returns the initialized parameters with a variable f_avg appended to it. This newly attached variable represents the average of the 50 final objective function values returned from the 10 different low-level PSO implementations at the specified parameters. A lower f_avg therefore corresponds to a set of parameters that achieved a better average objective function score than a higher average objective function score. Note that the termination criterion for the low-level, (and high-level as will be seen later) PSO algorithms are both allowed to search for k_{max} iterations without noticing an improvement and terminating the algorithm. The termination criterion is therefore held constant across the two algorithms.

Now that the evaluation of performance for the high-level PSO has been discussed, I will continue to discuss the init_PSO function with the high-level option specified. I started this option by initializing the locally best found solutions, their respective velocities, and the actual parameters. The high-level option then enters a for-loop, which iterates for the defined amount of particles; in which it initializes ω, C_1, C_2, their corresponding locally best found solutions, and their velocities. After reviewing the literature, I found that the following ranges seem to be common for PSO algorithms: $0.4 \leq \omega \leq 0.9$, and $0.5 \leq C_1, C_2 \leq 2$; and the parameters were initialized to values within these ranges by drawing from a uniform distribution and rounding the values to one digit. After this initialization of local best and positions, and similarly to the low-level PSO; an initialization factor was applied to the positions to determine the initial starting velocities for the respective combinations of parameters. Thereafter, the parameters are merged and evaluated by evaluate_PSO, which as discussed previously returns the average of the objective function values from the low-level PSO. The set of parameters with the minimum average objective function value is then saved as the global best set. These global best variables are then appended to a dataframe and the function returns this dataframe as the initialization of the particles for the high-level PSO. At this point, I would like the reader to notice that the initialization variables and maximum iterations are currently left as variables that can be adjusted as global variables before executing the problem.

```r
# Function to assess PSO's -------------------------------------------------
evaluate_PSO <- function(parameters, k_max, option)
{
  # Input: Parameters, max iterations, option
  # Output: Option 1: Updated set of particles from single set of parameters
  # Output: Option 2: Non-updated set of parameters with average objective value appended
  if(option == "low_level") # Low-level PSO evaluation
  {
    t <- k <- 0 # Sets counter and iteration to zero
    particles <- init_PSO(num_partic, option = "low_level") # Initialize particles
    while(k < k_max) # While termination criterion not met
    {
      counter_pre <- unique(particles$best_f) # Pre-update: Best objective
      particles <- update_PSO(particles,
                              omega = parameters$omega, # Note: Single set of parameters
                              c_1 = parameters$c_1,
                              c_2 = parameters$c_2,
                              option = "low_level") # Update particles using single set
      counter_post <- unique(particles$best_f)  # Post-update: Best objective
      t <- t + 1 # Increment iterations
      ifelse(test = counter_pre == counter_post, # If no change in best objective
             yes =  k <- k + 1, # Increment no-change counter
             no = k <- 0) # Else reset no-change counter
    }
    return(particles) # Output: Updated set of particles
  }
  if(option == "high_level") # High-level PSO evaluation
  {
    avg_solns <- c() # Keep track of the avg objective found in low_level
    for(i in 1:nrow(parameters))  # For each particle (10)
    {
      t <- k <- 0 # Sets counter and iteration to zero
      particles <- init_PSO(num_partic, option = "low_level") # Initialize particles
      while(k < k_max) # While less than max iterations
      {
        counter_pre <- unique(particles$best_f) # Pre-update: Best objective
        particles <- update_PSO(particles,
                                omega = parameters$omega[i], # Note: Multiple sets of parameters
                                c_1 = parameters$c_1[i],
                                c_2 = parameters$c_2[i],
                                option = "low_level") # Update particles using single set
        counter_post <- unique(particles$best_f) # Post-update: Best objective
        t <- t + 1 # Increment iterations
        ifelse(test = counter_pre == counter_post, # If no change in best objective
               yes =  k <- k + 1, # Increment no-change counter
               no = k <- 0) # Else reset no-change counter
      }
      avg_solns[i] <- mean(particles$f) # Find the average objective function value achieved
    }
    parameters$f_avg <- avg_solns # Append average objective function values
    return(parameters) # Output: Parameters with average objective function values
  }
}
```

The evaluate_PSO function previously mentioned is displayed above. The function takes as input a set of parameters, the same maximum amount of iterations k_{max} previously discussed, and once again an option specifying whether this is a high- or low-level PSO evaluation. I would like the reader to note that this function contains the exact generic PSO template for the low-level swarm particles. For the low-level option, this function requires a single set of parameters corresponding to ω, C_1, and C_2; which the function evaluates until the stopping criterion corresponding to k_{max} is met, and the function returns the set of particles. The low-level option starts by initializing the iterations t, the counter k, and it initializes a low-level PSO with (num_partic = 50) particles. The function then enters a loop that checks the termination criterion by observing the change in the best objective function value found. Note that up to this point, there has been no discussion of updating the positions or velocities of the swarm. This will be performed by the function called update_PSO. This will be discussed later, but for now I would just like the reader to note that it takes as input the low-level swarm particles, a single set of parameters corresponding to ω, C_1, and C_2, and an option specifying which level of PSO is being considered. The update_PSO function then returns the set of particles with updated positions, velocities, local best found solutions, and the globally best found solution. This low-level option's purpose is therefore only to evaluate the optimal or near-optimal single set of parameters passed to it, upon a randomly initialized low-level PSO. After finishing the low-level PSO updating process, the function returns an updated set of particles.

The high-level option performs a similar purpose, but instead of evaluating a single set of parameters and returning the updated particles, the high-level option evaluates a set of parameters corresponding to the high-level swarm, and returns the non-updated set of parameters with the average objective function value achieved by the low-level PSO algorithms appended to it. The function starts by declaring a variable avg_solns which will be used to keep track of the average objective function values found by the low-level PSO. Then the function iterates for each parameter, and declares the counter and iteration and initializes a fresh set of low-level swarms for each parameter. I would like the reader to notice that it the low-level PSO is initialized randomly for each set of parameters, and the same initialization is not reused across iterations. I believe this will add to the general robust-ness of the algorithm. After initializing these particles for the set of parameters, the function enters a while loop which iterates until the previously mentioned termination criterion is met. This termination criterion was applied in an identical manner to the low-level PSO option, in which the function keeps a copy of the best objective function value within the swarm before and after the low-level PSO is updated. If these values are the same, the function increments the no-change counter otherwise it is reset to zero. If this no-change counter exceeds the pre-defined amount of iterations that are allowed to pass without noticing an improvement, the function enters the while loop and the average objective function value of the low-level particle is recorded. The function repeats this process for all 10 sets of parameters, after which the function returns the parameters with their appended average objective function values.

The update_PSO function will be considered next. This is quite a large function, and it performs the body of the for loop within the generic template for the PSO. In fact, the entire low-level PSO template can be seen within the above and below functions. update_PSO repeatedly applies the following two equations to each parameter.

$$v_i(t) = \omega v_i(t-1) + \rho_1 C_1[p_i - x_i(t-1)] + \rho_2 C_2[p_g - x_i(t-1)]$$
$$x_i(t) = x_i(t-1) + v_i(t)$$

The first equation relates to updating the velocity, or the magnitude of change that is applied to a certain parameter, and the second equation relates to updating the position by adding the calculated velocity to the previous position. It is important to note that prior to this point, the particles dataframe (i.e. the low-level swarm) contains the variables

$$x, y, v_x, v_y, p_{i_x}, p_{i_y}, f, p_{g_x}, p_{g_y}, p_{g_f}$$

Similarly, the parameters dataframe (i.e. the high-level swarm) contains the variables

$$\omega, C_1, C_2, v_\omega, v_{C1}, v_{C2}, p_{i_\omega}, p_{i_{C1}}, p_{i_{C2}}, p_{g_\omega}, p_{g_{C1}}, p_{g_{C2}}, p_{g_{favg}}$$

```r
# Update particles/parameters ---------------------------------------------
update_PSO <- function(particles, omega, c_1, c_2, option, parameters)
{
  # Input: Option: Low-level: Particles, omega, c_1, c_2, option = "low_level"
  # Output Option: Low-level: Updated set of particles
  if(option == "low_level") # Option: Low-level PSO selected
  {
    x_lim <- y_lim <- c(-512, 512) # Boundaries of search space
    for(i in 1:nrow(particles)) # For each particle (50)
    {
      rho_1 <- runif(1) # Random rho between [0,1]
      rho_2 <- runif(1) # Random rho between [0,1]
      # Update the x velocity
      particles$v_x[i] <- omega*particles$v_x[i] +
        rho_1*c_1*(particles$p_i_x[i] - particles$x[i]) +
        rho_2*c_2*(unique(particles$best_x) - particles$x[i])

      # Update the y velocity
      particles$v_y[i] <- omega*particles$v_y[i] +
        rho_1*c_1*(particles$p_i_y[i] - particles$y[i]) +
        rho_2*c_2*(unique(particles$best_y)- particles$y[i])

      # Update the x position
      particles$x[i] <- particles$x[i] + particles$v_x[i]
      particles$x[i] <- scales::squish(particles$x[i], x_lim) # Ensure within boundaries

      # Update the y position
      particles$y[i] <- particles$y[i] + particles$v_y[i]
      particles$y[i] <- scales::squish(particles$y[i], y_lim) # Ensure within boundaries

      # Evaluate the new objective value
      particles$f[i] <- evaluate_objective(particles$x[i], particles$y[i])

      # Accept the new points as locally best found if better than previous best found
      if(particles$f[i] < evaluate_objective(particles$p_i_x[i], particles$p_i_y[i]))
      {
        particles$p_i_x[i] <- particles$x[i] # Accept x as local best
        particles$p_i_y[i] <- particles$y[i] # Accept y as local best
      }
      # Accept the new points as globally best found if better than previous best found
      if(particles$f[i] < unique(particles$best_f))
      {
        particles$best_x <- particles$x[i] # Accept x as globally best
        particles$best_y <- particles$y[i] # Accept y as globally best
        particles$best_f <- particles$f[i] # Accept f as globally best
      }
    }
    return(particles) # Return updated set of particles
  }
```

```r
if(option == "high_level")
{
  # Input: Option: High-level: Parameters, omega, c_1, c_2, option = "high_level"
  # Output Option: High-level: Updated set of parameters
  c_lim <- c(0.5, 2) # Boundaries of search space
  omega_lim <- c(0.4, 0.9) # Boundaries of search space
  for(i in 1:nrow(parameters)) # For each particle
  {
    rho_1 <- runif(1) # Random rho between [0,1]
    rho_2 <- runif(1) # Random rho between [0,1]
    # Update omega velocity
    parameters$v_omega[i] <- omega*parameters$v_omega[i] +
      rho_1*c_1*(parameters$p_i_omega[i] - parameters$omega[i]) +
      rho_2*c_2*(unique(parameters$best_omega) - parameters$omega[i])

    # Update c_1 velocity
    parameters$v_c_1[i] <- omega*parameters$v_c_1[i] +
      rho_1*c_1*(parameters$p_i_c_1[i] - parameters$c_1[i]) +
      rho_2*c_2*(unique(parameters$best_c_1) - parameters$c_1[i])

    # Update c_2 velocity
    parameters$v_c_2[i] <- omega*parameters$v_c_2[i] +
      rho_1*c_1*(parameters$p_i_c_2[i] - parameters$c_2[i]) +
      rho_2*c_2*(unique(parameters$best_c_2) - parameters$c_2[i])

    # Update the omega position
    parameters$omega[i] <- parameters$omega[i] + parameters$v_omega[i]
    parameters$omega[i] <- scales::squish(parameters$omega[i], omega_lim) # Ensure within boundaries

    # Update the c_1 position
    parameters$c_1[i] <- parameters$c_1[i] + parameters$v_c_1[i]
    parameters$c_1[i] <- scales::squish(parameters$c_1[i], c_lim) # Ensure within boundaries

    # Update the c_2 position
    parameters$c_2[i] <- parameters$c_2[i] + parameters$v_c_2[i]
    parameters$c_2[i] <- scales::squish(parameters$c_2[i], c_lim) # Ensure within boundaries

    # Update average objective function value at new parameters
    parameters$f_avg[i] <- evaluate_PSO(parameters = parameters[i,],
                                        k_max = k_max,
                                        option = "high_level")$f_avg

    # Accept the new parameters as locally best found if better than previous best found
    if(parameters$f_avg[i] < evaluate_PSO(parameters = data.frame(omega = parameters$p_i_omega[i],
                                                                  c_1 = parameters$p_i_c_1[i],
                                                                  c_2 = parameters$p_i_c_2[i]),
                                          k_max = k_max, option = "high_level")$f_avg)
    {
      parameters$p_i_omega[i] <- parameters$omega[i]
      parameters$p_i_c_1[i] <- parameters$c_1[i]
      parameters$p_i_c_2[i] <- parameters$c_2[i]
    }
```

```
      # Accept the new parameters as globally best found if better than previous best found
      if(parameters$f_avg[i] < evaluate_PSO(data.frame(omega = unique(parameters$best_omega),
                                                       c_1 = unique(parameters$best_c_1),
                                                       c_2 = unique(parameters$best_c_2)),
                                            k_max = k_max, option = "high_level" )$f_avg)
      {
        parameters$best_omega <- parameters$omega[i]
        parameters$best_c_1 <- parameters$c_1[i]
        parameters$best_c_2 <- parameters$c_2[i]
      }
    }
    parameters <- round(parameters, digits = 1) # Round the digits to one decimal place
    return(parameters) # Return updated set of parameters
  }
}
```

The low-level PSO option is displayed above as the first option for the function. The function requires as input an existing set of the low-level swarm *particles* with the variables mentioned previously present within it. The function also requires a single set of parameters ω, C_1, and C_2 be passed to it, and the option type. Upon entering this low-level PSO option, the limits of the search space are declared and the function enters a for-loop for all the particles present within the particles variable (i.e. the low-level swarm). Two random variables are drawn from a uniform distribution and declared as $\rho_{1,2} = rand([0,1])$. The loop then continues and updates the (x, y) velocities, after which it updates the (x, y) variable positions. After each update of the (x, y) positions, I implemented the squish function from the scales library to ensure that the particles are not leaving the boundaries of the search space. Thereafter, the objective function is evaluated at the newly updated (x, y) points to update the variable f, and if the points are better (i.e. less; since this is a minimization problem) than the past locally best found solutions, then the current solutions are accepted as the locally best found solutions of the specific particle within the low-level swarm, particles. In a similar fashion, if the evaluated objective function is better than the previous globally best found objective value p_{g_f} (which the unique function is applied to to remove duplicate entries within the dataframe), then the variables p_{g_x}, p_{g_y}, and p_{g_f} are updated by accepting the current solution as the new globally best found solution. This process is repeated for all the entries within the swarm, and the function returns the updated swarm (i.e. particles) after all the particles have been updated .

The high-level PSO option is displayed second, and this function performs a nearly identical process: it takes as input the high level swarm (i.e. parameters) which contains 10 different combinations of parameters for the low-level PSO to be tested on. The function also takes as input three variables (ω, C_1, C_2) which will be used for the high level PSO. These variables are specified to be $(1, 1, 1)$ in the assignment detail. Note the search space is no longer defined by (x, y) but now by (ω, C_1, C_2). The function starts by defining the limits of the search space which were previously established as $0.5 \leq C_1, C_2 \leq 2$ and $0.4 \leq \omega \leq 0.9$. The function then enters a loop which iterates through each set of parameters within the swarm (which I named parameters), and declares per loop two random values to introduce some stochastic nature to the update equations, i.e. $\rho_{1,2} = rand([0,1])$. Thereafter, the velocity of the parameters (ω, C_1, C_2) are updated by using exactly the same equations as before with the low-level update, but the equations are altered to operate on the variables present within the parameters dataframe. Once again, after the positions are updated, I used the squish function to cap maximum and minimum values at the search space boundaries. The new updated position of the parameters are evaluated and a new average objective function value is appended by using the evaluate_PSO function. If this average objective function value is less than the previous best found local solution's average objective function value, then the set of parameters are accepted as the new best found local solution. Similarly, if this average objective function value is less than the previous globally best found solution, then it is accepted as the new best found global solution. After the entire high-level swarm has been iterated through, the swarm is rounded to one decimal place and returned from the function as the updated set of parameters.

List of functions for Question 1

- evaluate_objective()
  - Options: None
  - Input: Set of decision variables
  - Output: Calculated objective value(s)
  - Dependencies: None
- plot_search()
  - Options: 2
  - Input 1: Viewing angles (theta, phi), and option specified as "space"
  - Output 1: Graph showing decision space
  - Input 2: Viewing angles (theta, phi), and option specified as "swarm"
  - Output 2: Graph showing decision space with low-level PSO final particles illustrated
  - Dependencies: evaluate_PSO, evaluate_objective
- init_PSO()
  - Options: 2
  - Input 1: Number of particles within swarm, and option specified as "low_level"
  - Output 1: Initialized particles for low-level PSO
  - Input 2: Number of particles within swarm, and option specified as "high_level"
  - Output 2: Initialized parameters for low-level PSO
  - Dependencies: evaluate_PSO, evaluate_objective
- evaluate_PSO()
  - Options: 2
  - Input 1: Single set of parameters to test low-level PSO, max iterations before termination, option specified as "low_level"
  - Output 1: Updated set of particles that illustrates how the low-level algorithm performed at the specified parameters.
  - Input 2: Multiple sets of parameters to assess high-level PSO, max iterations before termination, option specified as "high_level"
  - Output 2: Non-updated set of parameters with average objective value appended to original set of parameters
  - Dependencies: init_PSO, evaluate_PSO
- update_PSO()
  - Options: 2
  - Input 1: Low-level particles to update, parameters (ω, C_1, C_2) to use with the low-level particles, and an option specified as "low_level"
  - Output 1: Updated positions, velocities, locally and globally best found solutions of low-level particles.
  - Input 2: High-level parameters to update, parameters (ω, C_1, C_2) to use with the high-level updating, and an option specified as "high_level"
  - Output 2: Updated positions, velocities, locally and globally best found solutions of high-level parameters.
  - Dependencies: evaluate_PSO, evaluate_objective
- High-level parameters to update, k_{max}, option
  - Output 1: Updated set of particles
  - Output 2: Updated set of parameters
  - Dependencies: evaluate_PSO

These functions will be used in conjunction with a hybrid function that I called hybrid_PSO. This function will call the above functions, which will utilize the low-level PSO to solve the decision space. This hybrid function is the same as running one single PSO algorithm. Therefore prior to illustrating the implementation, I would like to briefly recap on the generic template for the PSO algorithm that was provided to us in class. The algorithm takes as input an inertia ω, C_1, and C_2 and it outputs an approximate solution to the respective optimization problem.

1) t = 0, random initialization of swarm's position's, and velocities.
2) Initialize p_i for each particle i and p_g for the entire swarm of particles
3) Increment counter t = t + 1
4) For each particle within the swarm:
5) Update the velocity by using the velocity-update equation previously specified
6) Update the position by using the position-update equation previously specified
7) Update the local best if the updated particle is better than the previous locally best found particle
8) Update the global best if the updated particle is better than the previous globally best found particle
9) If stopping criterion is met, break from for loop, output p_g and terminate algorithm
10) If stopping criterion is not met, repeat from Step 3

```
hybrid_PSO <- function(omega_high, c_1_high, c_2_high)
{ # Input (omega, C_1, C_2) for high-level PSO
  t <- k <-  0 # Step 1: Initialize counter
  parameters <- init_PSO(num_param, option = "high_level") # Step 1 & 2: Initialize swarm, p_g, p_i
  while(k < k_max) # Step 9 & 10: While stopping criterion is not met
  {
    t <- t + 1 # Step 3: Increment counter

    # Counter: Pre-update
    counter_pre <- unique(parameters[, c("best_omega", "best_c_1", "best_c_2")])

    parameters <- update_PSO(parameters = parameters, # Steps 4 & 8: Update all particles
                             option = "high_level", # Steps 4 & 8: Update position
                             omega = omega_high, # Steps 4 & 8: Update velocity
                             c_1 = c_1_high, # Steps 4 & 8: Update local best
                             c_2 = c_2_high) # Steps 4 & 8: Update global best
    # Counter: Post-update
    counter_post <- unique(parameters[, c("best_omega", "best_c_1", "best_c_2")])

    ifelse(test = ((counter_pre$best_omega == counter_post$best_omega) && # Check pre- and post-counter
                   (counter_pre$best_c_1 == counter_post$best_c_1) && # Global best found solns
                   (counter_pre$best_c_2 == counter_post$best_c_2)),  # If they are the same
           yes = k <- k + 1, # Increment no-change counter
           no = k <- 0)  # Else, reset no-change counter
  }
  names(counter_post) <- c("omega", "c_1", "c_2") # Renaming of variables
  return(list(counter_post, t)) # Return list with the counter
}
```

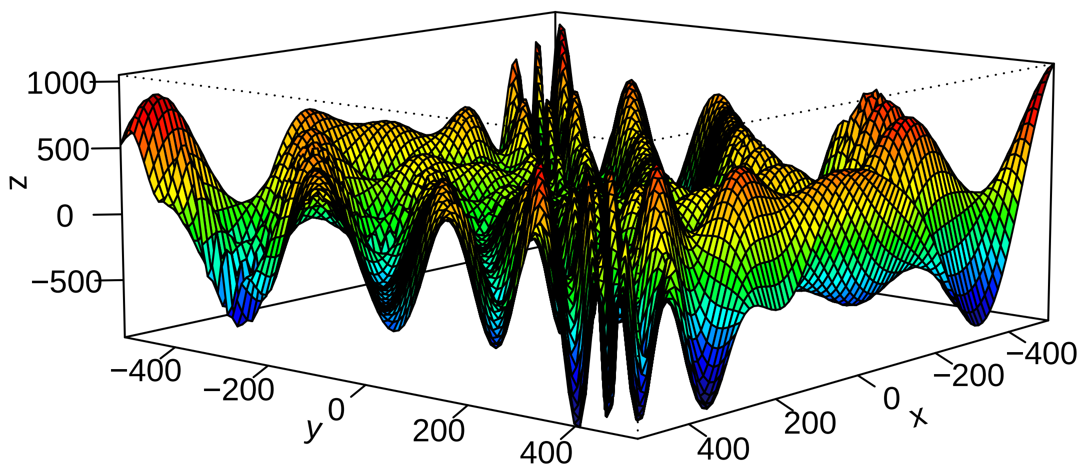
The function I implemented is displayed above. I included steps, if the reader wishes to compare this function to that of the generic PSO template provided to us. In fact, I would appreciate it if the reader could refer back to evaluate_PSO and notice the similarity between hybrid_PSO and evaluate_PSO. hybrid_PSO operates on the high-level PSO, and evaluate_PSO operates on the low-level PSO, but they both follow the exact same template as displayed above. hybrid_PSO starts by initializing the counter t, and the no-change counter k, to 0. Thereafter, it initializes a set of parameters (recall: parameters refers to the high-level swarm).

This init_PSO function has an internal call to evaluate_PSO to evaluate the low-level PSO algorithm at the initialized parameters, and init_PSO returns a initialized swarm with the positions, velocities, local best, average objective function values acquired by the low-level PSO, and the global best positions of the parameters within it. These variables are stored within the high-level swarm, and the hybrid_PSO function enters the while loop that has as a criterion that the no-change counter be smaller than the maximum amount of iterations that are allowed to pass without observing an change in the best-found parameter. This is set up by storing the global best positions prior to the update $(\omega_{best}, C_{1best}, C_{2best})$, and updating the position high-level PSO, and then storing the global best positions after the update $(\omega_{best}, C_{1best}, C_{2best})$. The update_PSO function between these two no-change counters updates the positions, velocities, local best, average objective function values acquired by the low-level PSO, and the global best positions of the parameters within it. update_PSO with the high-level option specified has an internal call to evaluate_PSO with the high-level option specified, and an entire low-level PSO algorithm is executed from this function call at the set of parameters, which initializes and updates a low-level swarm until termination. The updated set of parameters is returned, and the counter is checked after the update. If the counter has not changed, the no-change counter is incremented, and if it has changed the no-change counter is reset. Eventually this stopping criterion is met, and as specified in the generic PSO template, the function outputs the global best found solution (which is contained within the prior- and post-update counters). The variables are renamed to remove the "best_" part that trails them, and the function returns the globally best found solution, and the amount of iterations (which is not part of the generic PSO template but I wished to inspect it for the sake of this report).

Investigate performance

The performance of the implemented functions will be investigated now. Prior to starting this discussion, let's visualize the search space by using the plot_search function declared earlier.

```
plot_search(phi = 10, theta = 130, option = "space") # Investigate decision space
```



The function has quite a few local optimum, but the true global optimum can be seen at the front of the 3D graph at approximately the maximum positive x-value and around 400 for the y-value. Let's apply the hybrid-PSO to the search space. I started by declaring some global variables for the functions which can be varied to investigate the algorithms change in performance. The first of these variables is the maximum iterations that are allowed to pass without noticing any change in the global best found solution, k_max. Thereafter, I set the variable num_param, which reflects the amount of particles that are used within the high-level swarm; similarly I initialized num_partic, which reflects the amount of particles that are used within the low-level swarm. Thereafter, I declared two variables init_low and init_high which are responsible for initializing the velocities as a certain factor of the position. Similarly, the inertia and cognitive factors for the high-level PSO were initialized to one as the assignment specified. I noticed that the velocity initialization has some effect on the speed of convergence. The initialization factors were kept the same for both low and high-level PSO's, and upon testing and I found that for initialization factors corresponding to (0, 0.1, 0.5, 1) the algorithm terminated within (17, 15, 25, 11) iterations respectively. I decided to proceed with the latter option.

```
# Global parameters -------------------------------------------------------
k_max <- seed_val <-  5 # Maximum iterations before terminating PSO's
num_param <- 10 # Swarm size: High-level PSO
num_partic <- 50 # Swarm size: Low-level PSO
init_low <- 1 #  Velocity initialization factor: Low-level PSO
```

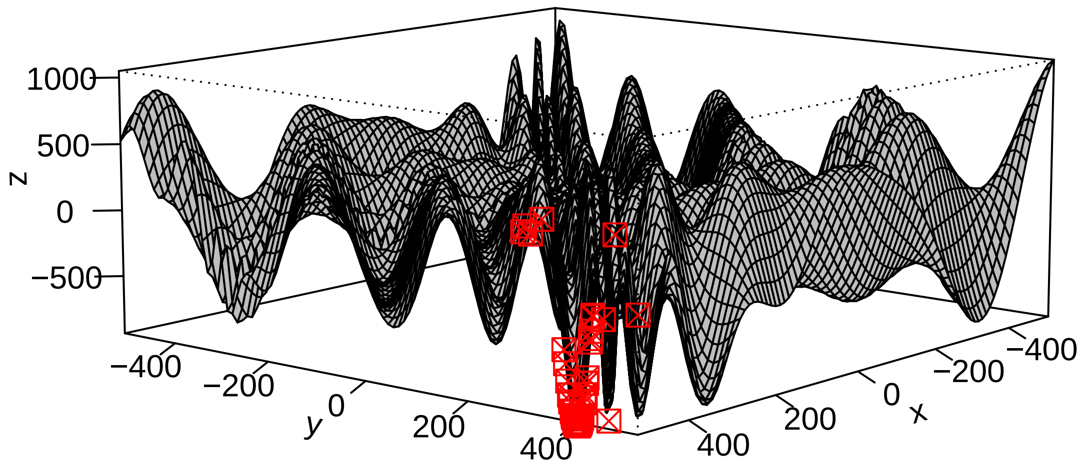
```r
init_high <- 1 # Velocity initialization factor: High-level PSO
omega_high <- 1 # High-level PSO: Inertia
c_1_high <- 1 # High-level PSO: Cognitive factor 1
c_2_high <- 1 # High-level PSO: Cognitive factor 2
```

```r
# Hybrid PSO execution ---------------------------------------------------
set.seed(seed_val) # Reproducible results
results <- hybrid_PSO(omega_high, c_1_high, c_2_high) # Execute and store results
results # Investigate results
```

```
## [[1]]
##   omega c_1 c_2
## 1   0.4 0.5   2
##
## [[2]]
## [1] 11
```

The optimal parameters for this execution are displayed above. I did notice that the optimal parameters could change for new executions of the hybrid algorithm, and that by using the same seed, increasing k_max higher did not change the output of the hybrid PSO. Upon reviewing the literature, I believe this makes sense since this is a stochastic algorithm and there are a multitude of things that can change with each run, and since the search space is relatively small for the high-level PSO not so many iterations are required. I proceeded to use these parameters to investigate how a low-level PSO algorithm would perform on the search space. For the purpose of my report, I made a slight change in my code for the plot_search function where I just removed the velocity from the particles dataframe before returning it. This is simply to illustrate the final set of particles for the low-level PSO better, since the inclusion of the velocities provided some formatting issues. I did note that the closer that the particles move to the global optimum, the smaller their velocities become. I reran the particles with the set of parameters with 3 different limits on the stopping criterion, corresponding to $k_{max} = (2, 5, 10)$. The first test corresponding to a k_{max} of 2 is displayed below.

```r
set.seed(seed_val) # Reproducible results
k_max <- 2 # Maximum iterations before terminating low-level PSO
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision
```

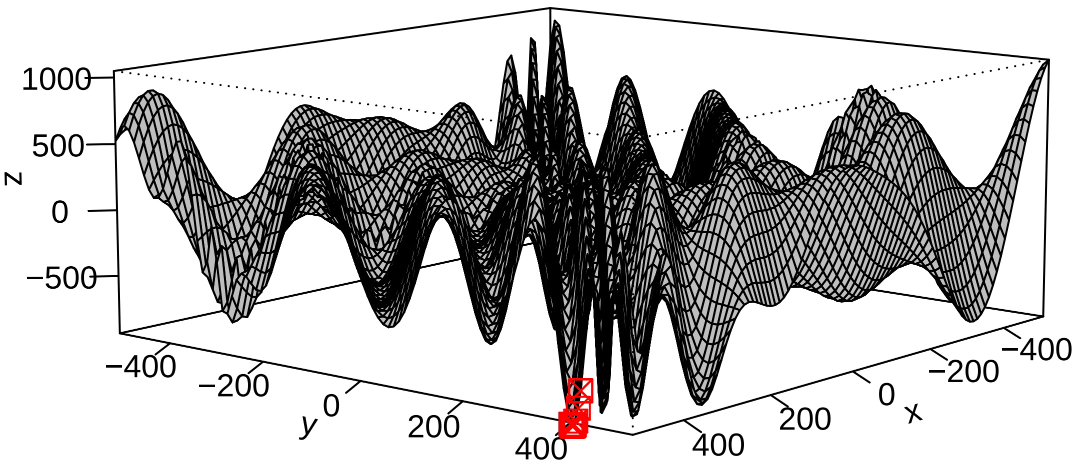



```
##            x        y      p_i_x       p_i_y         f best_x  best_y    best_f
## 1  512.0000 431.6622  512.0000 467.208686 -191.0380    512 404.487 -959.5665
## 2  512.0000 391.7189  512.0000 391.718855 -802.1769    512 404.487 -959.5665
## 3  512.0000 405.3021  512.0000 405.302088 -958.3295    512 404.487 -959.5665
## 4  512.0000 399.9109  512.0000 401.863749 -939.1207    512 404.487 -959.5665
## 5  512.0000 394.6109  512.0000 394.610927 -863.2562    512 404.487 -959.5665
## 6  512.0000 400.1782  512.0000 401.658677 -941.5380    512 404.487 -959.5665
## 7  512.0000 396.7995  512.0000 396.799505 -900.7292    512 404.487 -959.5665
## 8  512.0000 421.5140  512.0000 414.363973 -614.6774    512 404.487 -959.5665
## 9  512.0000 318.7282 -115.1736 397.178646  304.8641    512 404.487 -959.5665
## 10 512.0000 339.7206  358.2834 486.675311  417.9821    512 404.487 -959.5665
## 11 512.0000 399.6943  512.0000 406.065971 -937.0553    512 404.487 -959.5665
## 12 512.0000 420.4458  512.0000 404.487050 -654.1943    512 404.487 -959.5665
## 13 512.0000 512.0000 -442.9398    6.823886 -126.1679    512 404.487 -959.5665
## 14 512.0000 410.9920  512.0000 410.991968 -906.0229    512 404.487 -959.5665
## 15 471.4827 426.9303  471.4827 426.930303 -944.4371    512 404.487 -959.5665
## 16 512.0000 382.5399  512.0000 382.539852 -545.2937    512 404.487 -959.5665
## 17 492.4833 287.6739  349.2071 226.437971  289.0545    512 404.487 -959.5665
## 18 512.0000 406.1883  512.0000 406.188305 -955.2367    512 404.487 -959.5665
## 19 512.0000 389.8857  512.0000 391.598178 -757.5447    512 404.487 -959.5665
## 20 512.0000 412.2890  512.0000 400.579575 -883.2668    512 404.487 -959.5665
```

```
## 21 512.0000 416.2356  512.0000 416.235567 -790.0012    512 404.487 -959.5665
## 22 512.0000 411.9743  512.0000 404.960994 -889.1553    512 404.487 -959.5665
## 23 512.0000 418.1371  512.0000 401.651762 -732.9177    512 404.487 -959.5665
## 24 512.0000 411.2036  512.0000 411.203601 -902.5835    512 404.487 -959.5665
## 25 512.0000 428.1246  512.0000 428.124603 -341.5220    512 404.487 -959.5665
## 26 512.0000 389.7494  512.0000 389.749382 -754.0637    512 404.487 -959.5665
## 27 512.0000 397.2909  512.0000 397.290863 -908.0004    512 404.487 -959.5665
## 28 512.0000 420.5452  512.0000 389.523361 -650.5931    512 404.487 -959.5665
## 29 512.0000 431.9402  512.0000 410.220007 -179.6167    512 404.487 -959.5665
## 30 512.0000 427.5687  512.0000 389.168983 -365.5181    512 404.487 -959.5665
## 31 494.8427 293.5233  466.8795 225.768734  334.5035    512 404.487 -959.5665
## 32 512.0000 406.9182  512.0000 406.918192 -951.3055    512 404.487 -959.5665
## 33 511.8025 396.2267  509.8864 402.236256 -893.8772    512 404.487 -959.5665
## 34 503.1799 443.3756  494.2902 447.896944 -208.8943    512 404.487 -959.5665
## 35 512.0000 408.0325  512.0000 407.020187 -942.8673    512 404.487 -959.5665
## 36 512.0000 399.2919  512.0000 408.710490 -932.9700    512 404.487 -959.5665
## 37 512.0000 401.6626  512.0000 401.662614 -952.2772    512 404.487 -959.5665
## 38 512.0000 403.2419  512.0000 403.241850 -958.5342    512 404.487 -959.5665
## 39 512.0000 408.5722  512.0000 401.704802 -937.7153    512 404.487 -959.5665
## 40 512.0000 379.5924  512.0000 406.668206 -450.7627    512 404.487 -959.5665
## 41 512.0000 395.0258  512.0000 400.864137 -870.9736    512 404.487 -959.5665
## 42 512.0000 409.6280  512.0000 409.628029 -925.6152    512 404.487 -959.5665
## 43 512.0000 429.2909  512.0000 389.674784 -291.2472    512 404.487 -959.5665
## 44 512.0000 403.2010  512.0000 403.201031 -958.4414    512 404.487 -959.5665
## 45 311.3607 305.0497 -180.2338 114.162121  151.7958    512 404.487 -959.5665
## 46 512.0000 399.7941  512.0000 405.301701 -938.0183    512 404.487 -959.5665
## 47 512.0000 386.3973  512.0000 396.240171 -662.3008    512 404.487 -959.5665
## 48 512.0000 400.3038  512.0000 400.303844 -942.6240    512 404.487 -959.5665
## 49 512.0000 399.6379  512.0000 406.877256 -936.5027    512 404.487 -959.5665
## 50 512.0000 417.4738  512.0000 411.227780 -753.6692    512 404.487 -959.5665
```

With a k_{max} of 2, the low-level PSO does not entirely converge. This might have been expected, if we realize that 2 iterations is quite a strict stopping criterion and it is not enough time for thorough exploration of the search space and convergence. However, even with this strict stopping criterion, the algorithm performed remarkably well and came very close to the global minimum of $f(512, 404.2319) = -959.6407$. I expect that if I provide a more sensible stopping criterion, the particles will move closer to the global minimum. The low-level PSO is executed with a k_{max} of 5 on the next page.

```r
set.seed(seed_val) # Reproducible results
k_max <- 5 # Maximum iterations before terminating low-level PSO
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision s
```

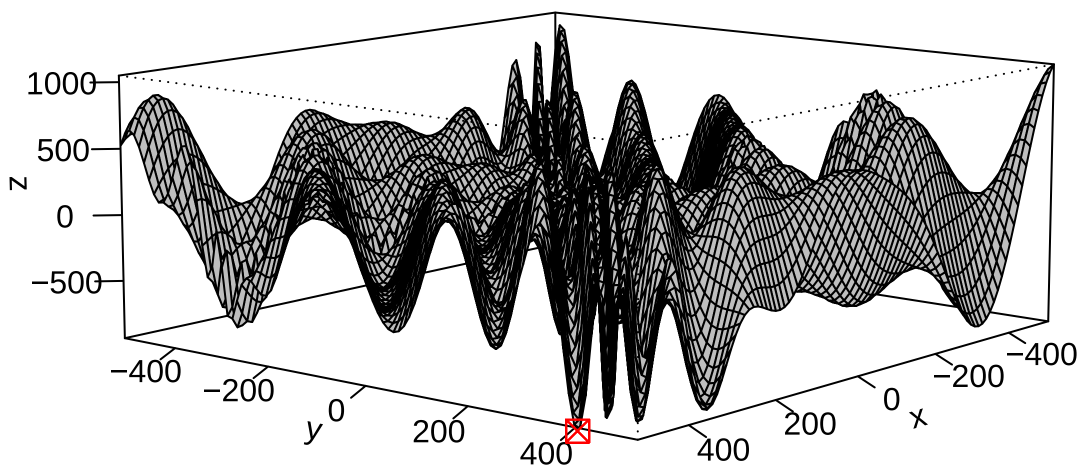



```
##      x        y p_i_x    p_i_y         f best_x  best_y    best_f
## 1  512 402.8269   512 403.9160 -957.4189    512 404.234 -959.6407
## 2  512 404.2346   512 404.2346 -959.6407    512 404.234 -959.6407
## 3  512 404.2980   512 404.2980 -959.6357    512 404.234 -959.6407
## 4  512 403.9494   512 403.9494 -959.5502    512 404.234 -959.6407
## 5  512 404.5497   512 404.2511 -959.5255    512 404.234 -959.6407
## 6  512 404.2410   512 404.2410 -959.6406    512 404.234 -959.6407
## 7  512 403.9030   512 404.4352 -959.5180    512 404.234 -959.6407
## 8  512 409.9872   512 407.5447 -920.8889    512 404.234 -959.6407
## 9  512 414.7634   512 414.7634 -828.9140    512 404.234 -959.6407
## 10 512 418.9335   512 409.9205 -706.8656    512 404.234 -959.6407
## 11 512 404.4779   512 404.1131 -959.5717    512 404.234 -959.6407
## 12 512 404.2347   512 404.2347 -959.6407    512 404.234 -959.6407
## 13 512 401.4303   512 401.4303 -950.9016    512 404.234 -959.6407
## 14 512 404.2256   512 404.2340 -959.6406    512 404.234 -959.6407
## 15 512 403.9659   512 403.9659 -959.5604    512 404.234 -959.6407
## 16 512 405.2677   512 405.2677 -958.4127    512 404.234 -959.6407
## 17 512 419.0149   512 403.9066 -704.1356    512 404.234 -959.6407
## 18 512 404.0118   512 404.0118 -959.5857    512 404.234 -959.6407
## 19 512 404.1149   512 404.2059 -959.6251    512 404.234 -959.6407
## 20 512 404.4917   512 404.4917 -959.5637    512 404.234 -959.6407
```

```
## 21 512 403.4658   512 404.2408 -958.9772    512 404.234 -959.6407
## 22 512 404.3422   512 404.2842 -959.6268    512 404.234 -959.6407
## 23 512 404.2788   512 404.2788 -959.6381    512 404.234 -959.6407
## 24 512 405.9378   512 405.2065 -956.2970    512 404.234 -959.6407
## 25 512 404.8089   512 403.8927 -959.2607    512 404.234 -959.6407
## 26 512 403.7893   512 403.7893 -959.4187    512 404.234 -959.6407
## 27 512 401.0871   512 403.1596 -948.6610    512 404.234 -959.6407
## 28 512 404.5451   512 404.3583 -959.5289    512 404.234 -959.6407
## 29 512 404.9045   512 404.3691 -959.1240    512 404.234 -959.6407
## 30 512 404.7861   512 404.3302 -959.2901    512 404.234 -959.6407
## 31 512 403.4701   512 404.5383 -958.9846    512 404.234 -959.6407
## 32 512 403.9890   512 404.0494 -959.5738    512 404.234 -959.6407
## 33 512 404.7162   512 404.4640 -959.3731    512 404.234 -959.6407
## 34 512 407.3034   512 406.9858 -948.7230    512 404.234 -959.6407
## 35 512 404.4597   512 404.4597 -959.5815    512 404.234 -959.6407
## 36 512 404.4068   512 404.1805 -959.6058    512 404.234 -959.6407
## 37 512 404.0305   512 404.1557 -959.5946    512 404.234 -959.6407
## 38 512 404.1791   512 404.1791 -959.6375    512 404.234 -959.6407
## 39 512 403.9185   512 404.3335 -959.5293    512 404.234 -959.6407
## 40 512 402.9901   512 403.0088 -957.9030    512 404.234 -959.6407
## 41 512 404.1992   512 404.2361 -959.6395    512 404.234 -959.6407
## 42 512 404.0819   512 404.1275 -959.6152    512 404.234 -959.6407
## 43 512 404.2686   512 404.2686 -959.6391    512 404.234 -959.6407
## 44 512 404.1620   512 404.1620 -959.6351    512 404.234 -959.6407
## 45 512 407.1466   512 402.7935 -949.8165    512 404.234 -959.6407
## 46 512 404.2550   512 404.2456 -959.6401    512 404.234 -959.6407
## 47 512 405.5250   512 404.6040 -957.7238    512 404.234 -959.6407
## 48 512 404.4604   512 404.3243 -959.5812    512 404.234 -959.6407
## 49 512 404.8109   512 404.1675 -959.2580    512 404.234 -959.6407
## 50 512 404.0262   512 404.0262 -959.5926    512 404.234 -959.6407
```

The graph above reveals that the algorithm is behaving as expected, and the particles of the low-level PSO are swarming towards the global optimum. All the x-points are at the global optimum value for x, but the y-value does not seem to exhibit full convergence yet. Let's increase the maximum iterations further and see what happens. I expect the swarm to be very close or at full convergence then. The low-level PSO is executed with a k_{max} of 10 on the next page.

```r
set.seed(seed_val) # Reproducible results
k_max <- 10 # Maximum iterations before terminating low-level PSO
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision
```




```
##       x        y p_i_x     p_i_y         f best_x   best_y    best_f
## 1   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 2   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 3   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 4   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 5   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 6   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 7   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 8   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 9   512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 10  512 404.2317   512 404.2318 -959.6407    512 404.2318 -959.6407
## 11  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 12  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 13  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 14  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 15  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 16  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 17  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 18  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 19  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 20  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
```

```
## 21 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 22 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 23 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 24 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 25 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 26 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 27 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 28 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 29 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 30 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 31 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 32 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 33 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 34 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 35 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 36 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 37 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 38 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 39 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 40 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 41 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 42 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 43 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 44 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 45 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 46 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 47 512 404.2319   512 404.2317 -959.6407    512 404.2318 -959.6407
## 48 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 49 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 50 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
```

That seems more like it! The algorithm is very close to fully converged. Entries 10 and 47 within the set of returned particles from the low-level swarm are the only two particles that have not converged to the global optimum. I noted that the global optimum was provided to us as $f(512, 404.2319) = -959.6407$, yet my algorithm seemed to find the global optimum at $f(512, 404.2318) = -959.6407$. This can be confirmed by inspecting particle 47 within the returned dataframe of the low-level swarm. This particle was at a y-value of 404.2319, yet it did not improve the objective function and it's respective locally best found solution remained at 404.2317. Similarly, particle 10 was at a y-value of 404.2317 but it's locally best found solution remained at 404.2318. It would therefore appear that my algorithm considers 404.2318 the best, 404.2317 the second best, and 404.2319 the third best. I believe that this could potentially be a rounding error on the assignment side, and I left the solution as is. I then proceeded to implement the multi-start local search algorithm.

High-level Relay Hybridisation: Multi-Start Local Search

A multi-start local search was implemented as a high-level relay hybridisation to improve upon five different non-optimal particles within the low-level PSO. The implementation was relatively straightforward, requiring only one additional function call from the evaluate_PSO function and the the set up of two additional functions. The first additional function that was set up is a function that is responsible for generating neighbours for the multi-start local search. The function was defined as follows, where the neighbours are random perturbations of the x and y coordinates of the received x and y coordinates.

```
generate_neigh <- function(NI_particles)
{ # Input: Set of 5 non-improving particles from low-level PSO
  # Output: A pre-specified number of neighbours to a single solution
  x <- NI_particles$x + runif(num_neigh, min = -512, max = 512) # Generate x-coordinates of neighbours
  y <- NI_particles$y + runif(num_neigh, min = -512, max = 512) # Generate y-coordinates of neighbours
  x <- scales::squish(x, c(-512, 512)) # Ensure neighbours within decision space limits
  y <- scales::squish(y, c(-512, 512)) # Ensure neighbours within decision space limits
  f <- evaluate_objective(x, y) # Evaluate these new neighbours
  return(data.frame(x, y, f)) # Returns dataframe containing coordinates and obj. value of neighbours
}
```

After generating the neighbours, their (x, y) positions are compressed to ensure these neighbours do not leave the decision space. The function then returns a dataframe containing the (x, y) coordinates of the neighbours, and their respective objective function values. Now that the neighbourhood function is defined, I will proceed to discuss the multi-start local search algorithm. A multi-start local search template may be defined as follows to find one single incumbant solution from k initial solutions. This template will have to be repeated for all 5 different non-optimal points that are to be improved.

1) Generate k initial starting solutions
2) For all k initial starting solutions, repeat 2 - 7
3) Generate neighbours to the k-th solution
4) Accept the most improving neighbouring solution as the k-th starting solution
5) Terminate the iteration of the for-loop when no more improving neighbours exist
6) Else return to step 2
7) Terminate the for loop when all solutions have been iterated through
8) Using these various optimum solutions returned, output the best solution as the best found solution

I started this implementation by declaring the MS_local_search function, as seen below. This functions takes the low-level PSO swarm particles as input, and outputs a set of particles from which the 5 worst solutions are improved by replacing them with the best solutions returned from a multi-start local search algorithm. The local-search algorithm is executed 5 times, as dictated by the variable num_improve. I included steps if the reader wishes to compare the flow of the metaheuristic to the generic template provided above. After improving the five worst solutions, which have the highest objective function values, I included a piece of code from the update_PSO function which checks whether a new global or local minimum has been reached at the five improved particles. This is important, since the improvement is performed after the call to update_PSO and therefore a new local or global minimum might be found.

```r
MS_local_search <- function(particles)
{ # Input: Low-level PSO swarm
  # Output: Low-level PSO that contains 5 particles improved by multi-start local search
  for(i in 1:num_improve) # The amount of particles that have to be improved, in our case 5
  {
    diff_solns <- c() # Variable to store different solutions from different starting solutions
    NI_particles <- particles[!particles$f %in% particles$best_f,] # All non-optimal particles
    idx <- sample.int(nrow(NI_particles), k_local) # Index: k non-optimal starting positions
    starting_pos <- NI_particles[idx,]# Step 1: Generate k initial starting solutions
    for(j in 1:nrow(starting_pos)) # Step 2: For all k initial starting solutions
    { # Step 7: Terminate the for loop when all solutions have been iterated through
      prior <- 0 # Counter: Objective value prior to generating neighbour
      post <- Inf # Counter: Objective value after generating neighbour
      while(prior < post) # Termination criterion
      {
        prior <- starting_pos$f[j] # Termination criterion: Prior to acceptance
        neighbours <- generate_neigh(starting_pos[j,]) # Step 3: Generate neighbours
        best_neigh <- neighbours[which.min(neighbours$f),] # Step 4: Most improving neighbour
        starting_pos[j,]$x <- best_neigh$x # Step 4: Accept most improving neighbour
        starting_pos[j,]$y <- best_neigh$y # Step 4: Accept most improving neighbour
        starting_pos[j,]$f <- best_neigh$f # Step 4: Accept most improving neighbour
        post <- best_neigh$f # Termination criterion: After acceptance
      }
      diff_solns <- rbind(diff_solns, best_neigh) # Combine solutions from local search
    }
    # Step 8: Choose the best solution as the incumbant solution
    best_soln <- diff_solns[which.min(diff_solns$f),] # Step 8
    particles$x[which.max(particles$f)] <- best_soln$x # Step 8
    particles$y[which.max(particles$f)] <- best_soln$y # Step 8
    particles$f[which.max(particles$f)] <- best_soln$f # Step 8
  }
  for(k in 1:nrow(particles)) # Update local and global best from new found solutions
  {
    # Accept the new points as locally best found if better than previous best found
    if(particles$f[k] < evaluate_objective(particles$p_i_x[k], particles$p_i_y[k]))
    {
      particles$p_i_x[k] <- particles$x[k] # Accept x as local best
      particles$p_i_y[k] <- particles$y[k] # Accept y as local best
    }
    # Accept the new points as globally best found if better than previous best found
    if(particles$f[k] < unique(particles$best_f))
    {
      particles$best_x <- particles$x[k] # Accept x as globally best
      particles$best_y <- particles$y[k] # Accept y as globally best
      particles$best_f <- particles$f[k] # Accept f as globally best
    }
  }
  return(particles) # Return: Improved particles
}
```

After setting the above function up; the evaluate_PSO function, which is responsible for executing the low-level PSO, needed to be adapted to include the MLS implementation. The updated function is displayed below, and I would like the reader to note that there is only one single line that has been added to both options, where the low-level swarm called particles is updated by sending it to the MS_local_search function previously defined.

```
# Function to assess PSO's -------------------------------------------------
evaluate_PSO <- function(parameters, k_max, option)
{
  # Input: Parameters, max iterations, option
  # Output: Option 1: Updated set of particles from single set of parameters
  # Output: Option 2: Non-updated set of parameters with average objective value appended
  if(option == "low_level") # Low-level PSO evaluation
  {
    t <- k <- 0 # Sets counter and iteration to zero
    particles <- init_PSO(num_partic, option = "low_level") # Initialize particles
    while(k < k_max) # While termination criterion not met
    {
      counter_pre <- unique(particles$best_f) # Pre-update: Best objective
      particles <- update_PSO(particles,
                              omega = parameters$omega, # Note: Single set of parameters
                              c_1 = parameters$c_1,
                              c_2 = parameters$c_2,
                              option = "low_level") # Update particles using single set
      particles <- MS_local_search(particles) # Note: Change for Multi-start Local Search
      counter_post <- unique(particles$best_f)  # Post-update: Best objective
      t <- t + 1 # Increment iterations
      ifelse(test = counter_pre == counter_post, # If no change in best objective
             yes =  k <- k + 1, # Increment no-change counter
             no = k <- 0) # Else reset no-change counter
    }
    return(particles) # Output: Updated set of particles
  }
  if(option == "high_level") # High-level PSO evaluation
  {
    avg_solns <- c() # Keep track of the avg objective found in low_level
    for(i in 1:nrow(parameters))  # For each particle (10)
    {
      t <- k <- 0 # Sets counter and iteration to zero
      particles <- init_PSO(num_partic, option = "low_level") # Initialize particles
      while(k < k_max) # While less than max iterations
      {
        counter_pre <- unique(particles$best_f) # Pre-update: Best objective
        particles <- update_PSO(particles,
                                omega = parameters$omega[i], # Note: Multiple sets of parameters
                                c_1 = parameters$c_1[i],
                                c_2 = parameters$c_2[i],
                                option = "low_level") # Update particles using single set
        particles <- MS_local_search(particles) # Note: Change for Multi-start Local Search
        counter_post <- unique(particles$best_f) # Post-update: Best objective
        t <- t + 1 # Increment iterations
        ifelse(test = counter_pre == counter_post, # If no change in best objective
               yes =  k <- k + 1, # Increment no-change counter
               no = k <- 0) # Else reset no-change counter
```

```
    }
    avg_solns[i] <- mean(particles$f) # Find the average objective function value achieved
  }
  parameters$f_avg <- avg_solns # Append average objective function values
  return(parameters) # Output: Parameters with average objective function values
 }
}
```

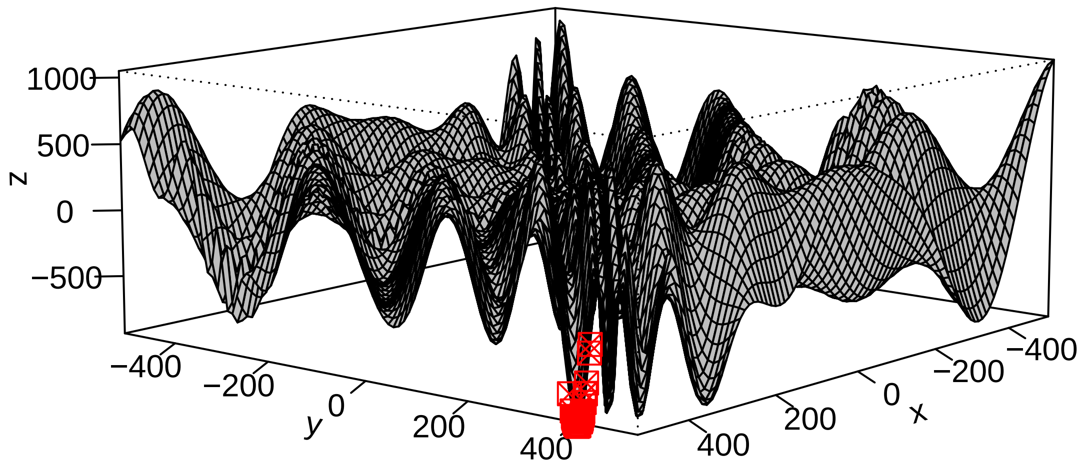
After updating the above function and including the MS_local_search and generate_neigh functions, all that remains to be performed is to investigate the performance of the algorithm. I started by declaring the required local variables as seen below. I chose 5 starting solutions arbitrarily as an initial value for this parameter. Similarly, I chose 1000 random perturbations to be applied to the local search positions to generate 1000 neighbors. Similarly to above, when calling the high-level PSO I kept the termination criterion at a $k_{max} = 5$.

```
num_improve <- 5 # Assignment specified: 5 solutions to improve
k_local <- 5 # 5 starting solutions
num_neigh <- 1000 # Amount of neighbouring solutions to consider
k_max <- 5 # Termination criterion
```

```
# High-level: HRH execution ----------------------------------------------
set.seed(seed_val) # Reproducible results
results <- hybrid_PSO(omega_high, c_1_high, c_2_high) # Execute and store results
```

As seen above, the parameters returned remain unchanged at the current seed and the algorithm converged in less iterations than previously. The MLS seems to be guiding the low-level swarm towards better solutions. I then proceeded to investigate the results of the low-level swarm with the newly updated functions as follows.

```r
set.seed(seed_val) # Reproducible results
k_max <- 2 # Termination criterion
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision
```

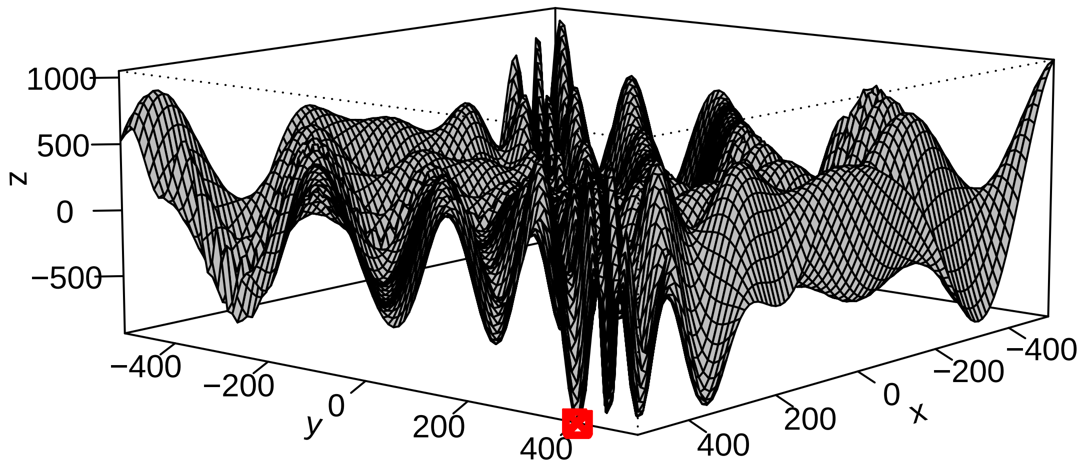



```
##       x        y p_i_x    p_i_y         f best_x   best_y    best_f
## 1   512 425.7828   512 404.1821 -442.0481    512 404.2331 -959.6407
## 2   512 404.2353   512 404.2353 -959.6406    512 404.2331 -959.6407
## 3   512 400.6682   512 403.6584 -945.5907    512 404.2331 -959.6407
## 4   512 412.6753   512 404.2915 -875.7207    512 404.2331 -959.6407
## 5   512 404.1945   512 404.2284 -959.6391    512 404.2331 -959.6407
## 6   512 401.0095   512 404.2591 -948.1195    512 404.2331 -959.6407
## 7   512 398.5679   512 404.3067 -924.8180    512 404.2331 -959.6407
## 8   512 401.2931   512 401.2931 -950.0356    512 404.2331 -959.6407
## 9   512 406.3150   512 404.4202 -954.6442    512 404.2331 -959.6407
## 10  512 404.8492   512 404.1256 -959.2056    512 404.2331 -959.6407
## 11  512 389.4809   512 410.5984 -747.1435    512 404.2331 -959.6407
## 12  512 418.4275   512 406.6456 -723.5580    512 404.2331 -959.6407
## 13  512 412.7019   512 404.9284 -875.1876    512 404.2331 -959.6407
## 14  512 401.0243   512 405.5050 -948.2234    512 404.2331 -959.6407
## 15  512 397.6788   512 404.4106 -913.4313    512 404.2331 -959.6407
## 16  512 399.3556   512 404.3809 -933.6379    512 404.2331 -959.6407
## 17  512 404.1516   512 404.1516 -959.6334    512 404.2331 -959.6407
## 18  512 412.4918   512 412.4918 -879.3498    512 404.2331 -959.6407
## 19  512 401.3416   512 405.2559 -950.3467    512 404.2331 -959.6407
## 20  512 405.7886   512 404.2836 -956.8588    512 404.2331 -959.6407
```

```
## 21 512 413.2895   512 403.9292 -862.9995    512 404.2331 -959.6407
## 22 512 409.5891   512 399.0866 -926.1090    512 404.2331 -959.6407
## 23 512 398.7443   512 399.8472 -926.8978    512 404.2331 -959.6407
## 24 512 427.1819   512 404.1595 -382.1869    512 404.2331 -959.6407
## 25 512 411.2142   512 397.3197 -902.4086    512 404.2331 -959.6407
## 26 512 420.5409   512 404.3664 -650.7517    512 404.2331 -959.6407
## 27 512 418.3146   512 404.2193 -727.2155    512 404.2331 -959.6407
## 28 512 394.5531   512 403.5036 -862.1584    512 404.2331 -959.6407
## 29 512 416.3200   512 416.3200 -787.6266    512 404.2331 -959.6407
## 30 512 406.3676   512 404.2005 -954.3876    512 404.2331 -959.6407
## 31 512 400.9825   512 400.9825 -947.9286    512 404.2331 -959.6407
## 32 512 404.5610   512 404.2382 -959.5172    512 404.2331 -959.6407
## 33 512 410.3880   512 405.1692 -915.2487    512 404.2331 -959.6407
## 34 512 404.1255   512 404.1255 -959.6278    512 404.2331 -959.6407
## 35 512 404.7210   512 404.7210 -959.3678    512 404.2331 -959.6407
## 36 512 397.2300   512 404.2389 -907.1228    512 404.2331 -959.6407
## 37 512 401.2142   512 401.2142 -949.5194    512 404.2331 -959.6407
## 38 512 403.3743   512 403.3743 -958.8096    512 404.2331 -959.6407
## 39 512 405.1405   512 403.9634 -958.6964    512 404.2331 -959.6407
## 40 512 404.0975   512 404.2331 -959.6202    512 404.2331 -959.6407
## 41 512 396.6544   512 401.3465 -898.4996    512 404.2331 -959.6407
## 42 512 406.3427   512 404.0492 -954.5096    512 404.2331 -959.6407
## 43 512 403.9004   512 404.3926 -959.5160    512 404.2331 -959.6407
## 44 512 404.3884   512 404.3884 -959.6128    512 404.2331 -959.6407
## 45 512 402.1669   512 402.1669 -954.8652    512 404.2331 -959.6407
## 46 512 405.6322   512 405.6322 -957.3916    512 404.2331 -959.6407
## 47 512 413.8077   512 413.8077 -851.5836    512 404.2331 -959.6407
## 48 512 402.7163   512 404.2837 -957.0576    512 404.2331 -959.6407
## 49 512 404.5953   512 404.3889 -959.4901    512 404.2331 -959.6407
## 50 512 402.1000   512 402.1000 -954.5533    512 404.2331 -959.6407
```

At a termination criterion of $k_{max} = 2$, this algorithm performed better than the initial hybrid PSO, as displayed on page 15. By comparing the two graphs for $k_{max} = 2$ we can see that the HRH algorithm is converging to the global optimum much faster. Let's investigate $k_{max} = 5$.

```r
set.seed(seed_val) # Reproducible results
k_max <- 5 # Termination criterion
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision
```

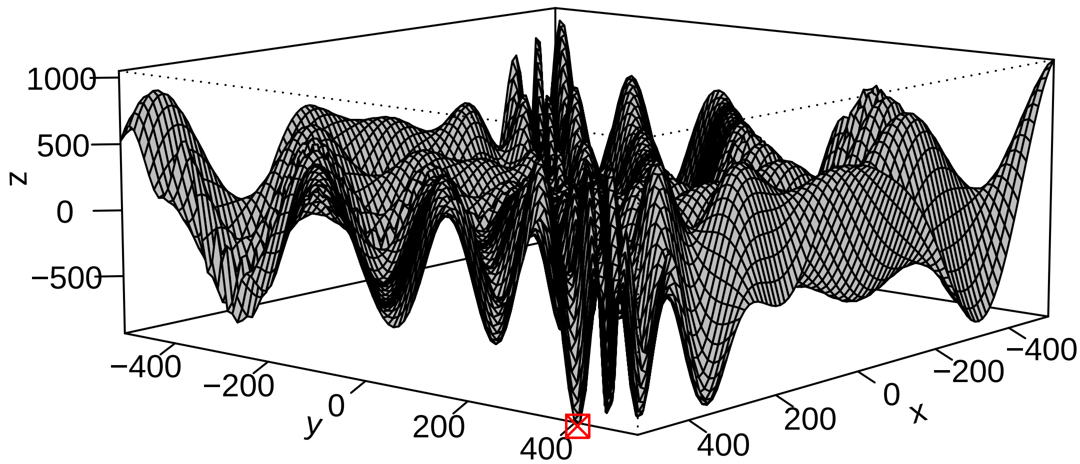



```
##      x        y p_i_x    p_i_y        f best_x   best_y    best_f
## 1  512 404.3532   512 404.1821 -959.6239    512 404.2331 -959.6407
## 2  512 403.9578   512 404.2353 -959.5555    512 404.2331 -959.6407
## 3  512 407.4319   512 403.6584 -947.7830    512 404.2331 -959.6407
## 4  512 402.5280   512 404.2915 -956.3803    512 404.2331 -959.6407
## 5  512 404.2021   512 404.2284 -959.6397    512 404.2331 -959.6407
## 6  512 403.8711   512 404.2591 -959.4931    512 404.2331 -959.6407
## 7  512 404.1622   512 404.1622 -959.6352    512 404.2331 -959.6407
## 8  512 408.0715   512 401.2931 -942.5181    512 404.2331 -959.6407
## 9  512 403.1548   512 404.4202 -958.3319    512 404.2331 -959.6407
## 10 512 404.0451   512 404.3157 -959.6011    512 404.2331 -959.6407
## 11 512 405.4251   512 403.9823 -958.0095    512 404.2331 -959.6407
## 12 512 399.6618   512 405.1091 -936.7379    512 404.2331 -959.6407
## 13 512 406.0477   512 404.2901 -955.8497    512 404.2331 -959.6407
## 14 512 404.3072   512 404.3072 -959.6342    512 404.2331 -959.6407
## 15 512 398.8221   512 404.4106 -927.7957    512 404.2331 -959.6407
## 16 512 408.8362   512 404.3809 -934.9407    512 404.2331 -959.6407
## 17 512 404.0175   512 404.1516 -959.5885    512 404.2331 -959.6407
## 18 512 404.4495   512 404.4495 -959.5867    512 404.2331 -959.6407
## 19 512 407.7391   512 405.2559 -945.3764    512 404.2331 -959.6407
## 20 512 404.3633   512 404.2836 -959.6210    512 404.2331 -959.6407
```

```
## 21 512 401.3488   512 403.9292 -950.3919    512 404.2331 -959.6407
## 22 512 399.1978   512 401.4321 -931.9685    512 404.2331 -959.6407
## 23 512 406.6940   512 401.9042 -952.6466    512 404.2331 -959.6407
## 24 512 404.9681   512 404.2544 -959.0215    512 404.2331 -959.6407
## 25 512 404.3693   512 404.3693 -959.6192    512 404.2331 -959.6407
## 26 512 403.9656   512 404.3664 -959.5603    512 404.2331 -959.6407
## 27 512 408.1377   512 404.2193 -941.9178    512 404.2331 -959.6407
## 28 512 404.3829   512 404.3829 -959.6147    512 404.2331 -959.6407
## 29 512 401.1430   512 406.3740 -949.0429    512 404.2331 -959.6407
## 30 512 404.2645   512 404.2005 -959.6394    512 404.2331 -959.6407
## 31 512 404.1412   512 404.1412 -959.6313    512 404.2331 -959.6407
## 32 512 404.1793   512 404.2382 -959.6375    512 404.2331 -959.6407
## 33 512 408.8408   512 405.1692 -934.8912    512 404.2331 -959.6407
## 34 512 404.5398   512 404.1255 -959.5326    512 404.2331 -959.6407
## 35 512 403.9327   512 404.3524 -959.5392    512 404.2331 -959.6407
## 36 512 404.2159   512 404.2389 -959.6404    512 404.2331 -959.6407
## 37 512 405.6956   512 405.6724 -957.1823    512 404.2331 -959.6407
## 38 512 402.5459   512 403.3743 -956.4480    512 404.2331 -959.6407
## 39 512 403.3858   512 404.1076 -958.8317    512 404.2331 -959.6407
## 40 512 404.7685   512 404.2331 -959.3121    512 404.2331 -959.6407
## 41 512 405.4950   512 405.4950 -957.8122    512 404.2331 -959.6407
## 42 512 409.1939   512 404.0492 -930.9139    512 404.2331 -959.6407
## 43 512 403.7949   512 404.3926 -959.4243    512 404.2331 -959.6407
## 44 512 405.2719   512 404.3884 -958.4027    512 404.2331 -959.6407
## 45 512 403.7502   512 403.7502 -959.3779    512 404.2331 -959.6407
## 46 512 400.5752   512 403.9638 -944.8590    512 404.2331 -959.6407
## 47 512 400.5318   512 403.9391 -944.5120    512 404.2331 -959.6407
## 48 512 404.3878   512 404.2837 -959.6130    512 404.2331 -959.6407
## 49 512 404.1789   512 404.2216 -959.6375    512 404.2331 -959.6407
## 50 512 405.4140   512 403.9327 -958.0399    512 404.2331 -959.6407
```

At a termination criterion of $k_{max} = 5$, this algorithm once again performed better than the initial hybrid PSO at the same termination criterion, as seen on page 17. The swarm is closer to convergence than the initial PSO algorithm was at this termination criterion. Let's investigate a termination criterion of $k_{max} = 10$.

```r
set.seed(seed_val) # Reproducible results
k_max <- 10 # Termination criterion
plot_search(parameters = results[[1]], phi = 10, theta = 130, option = "swarm") # Investigate decision
```




```
##       x        y p_i_x    p_i_y         f best_x   best_y    best_f
## 1  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 2  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 3  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 4  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 5  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 6  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 7  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 8  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 9  512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 10 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 11 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 12 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 13 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 14 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 15 512 404.1838   512 404.2327 -959.6380    512 404.2318 -959.6407
## 16 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 17 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 18 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 19 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 20 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
```

```
## 21 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 22 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 23 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 24 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 25 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 26 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 27 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 28 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 29 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 30 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 31 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 32 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 33 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 34 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 35 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 36 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 37 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 38 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 39 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 40 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 41 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 42 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 43 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 44 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 45 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 46 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 47 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 48 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 49 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
## 50 512 404.2318   512 404.2318 -959.6407    512 404.2318 -959.6407
```

We can recall that the initial algorithm did not fully converge for particle's 10 and 47 within the low-level swarm at a $k_{max} = 10$, as seen on page 19. By inspecting the swarm's locations, we can see that all swarm particles are now fully at the global optimum of $f(512, 404.2318)$, including particles' 10 and 47. Therefore, the MLS high-level relay hybridization is performing better than the initial algorithm in every way possible.

Conclusion

A hybrid-level PSO was implemented in this assignment, and its' performance was improved by adopting a high-level relay hybridization approach and using multi-start local search to improve upon five different non-optimal particles.