Optimización con algoritmo genético y Nelder-Mead

Joaquín Amat Rodrigo j.amatrodrigo@gmail.com

Febrero, 2019

Tabla de contenidos

Versión PDF: Github

Introducción

Los algoritmos genéticos son métodos de optimización heurística que, entre otras aplicaciones, pueden emplearse para encontrar el valor o valores que consiguen maximizar o minimizar una función. Su funcionamiento está inspirado en la teoría evolutiva de selección natural propuesta por Darwin y Alfred Russel: los individuos de una población se reproducen generando nuevos descendientes, cuyas características, son combinación de las características de los progenitores (más ciertas mutaciones). De todos ellos, únicamente los mejores individuos sobreviven y pueden reproducirse de nuevo, transmitiendo así sus características a las siguientes generaciones.

Los algoritmos genéticos son solo una de las muchas estrategias de optimización que existen, y no tiene por qué ser la más adecuada en todos los escenarios. Por ejemplo, si el problema en cuestión puede optimizarse de forma analítica, suele ser más adecuado resolverlo de esta forma.

La implementación de algoritmo genético que se muestra en este documento pretende ser lo más explicativa posible aunque para ello no sea la más eficiente.

El código de las funciones desarrolladas a lo largo del documento puede descargarse en el siguiente link.

Algoritmo

Aunque existen variaciones, algunas de las cuales se describen a lo largo de este documento, en términos generales, la estructura de un algoritmo genético para optimizar (maximizar o minimizar) una función con una o múltiples variables sigue los siguientes pasos:

---

1. Crear una población inicial aleatoria de *P* individuos. En este caso, cada individuo representa una combinación de valores de las variables.
2. Calcular la fortaleza (*fitness*) de cada individuo de la población. El *fitness* está relacionado con el valor de la función para cada individuo. Si se quiere maximizar, cuanto mayor sea el valor de la función para el individuo, mayor su *fitness*. En el caso de minimización, ocurre lo contrario.
3. Crear una nueva población vacía y repetir los siguientes pasos hasta que se hayan creado *P* nuevos individuos.

   3.1 Seleccionar dos individuos de la población existente, donde la probabilidad de selección es proporcional al *fitness* de los individuos.

   3.2 Cruzar los dos individuos seleccionados para generar un nuevo descendiente (*crossover*).

   3.3 Aplicar un proceso de mutación aleatorio sobre el nuevo individuo.

   3.4 Añadir el nuevo individuo a la nueva población.
4. Reemplazar la antigua población por la nueva.
5. Si no se cumple un criterio de parada, volver al paso 2.

En los siguientes apartados se describe cada una de las etapas del proceso para, finalmente, combinarlas todas en una única función.

Población inicial

En el contexto de algoritmos genéticos, el término individuo hace referencia a cada una de las posibles soluciones del problema que se quiere resolver. En el caso de maximización o minimización de una función, cada individuo representa una posible combinación de valores de las variables. Para representar dichas combinaciones, se pueden emplear vectores, cuya longitud es igual al número total de variables, y cada posición toma un valor numérico. Por ejemplo, supóngase que la función objetivo $J(x, y, z)$ depende de las variables x, y, z. El individuo $3{,}9.5, -0.5$, equivale a la combinación de valores $x = 3, y = 9.5, z = -0.5$.

El primer paso del algoritmo genético consiste en crear una población inicial aleatoria de individuos. La siguiente función crea una matriz en la que, cada fila, está formada por una combinación de valores numéricos aleatorios. Además, el valor para cada variable puede estar acotado dentro de un rango. Esta acotación resulta útil para agilizar el proceso de optimización, pero requiere disponer de información que permita acotar el intervalo de valores dentro del cual se encuentra la solución óptima.

```
crear_poblacion <- function(n_poblacion, n_variables, limite_inf = NULL,
                            limite_sup = NULL, verbose = TRUE) {

  # Esta función crea una matriz en la que, cada fila, está formada por una
  # combinación de valores numéricos aleatorios. El rango de posibles valores
  # para cada variable puede estar acotado.
  #
  # ARGUMENTOS
  # ============================================================================
  # n_poblacion: número total de individuos de la población.
  # n_variables: longitud de los individuos.
  # limite_inf:  vector con el límite inferior de cada variable. Si solo se
  #              quiere imponer límites a algunas variables, emplear NA para
  #              las que no se quiere acotar.
  # limite_sup:  vector con el límite superior de cada variable. Si solo se
  #              quiere imponer límites a algunas variables, emplear NA para
  #              las que no se quieren acotar.
  # verbose:     mostrar información del proceso por pantalla.
  #
  # RETORNO
  # ============================================================================
  # Una matriz de tamaño n_poblacion x n_variables que representa una población.

  # Comprobaciones
  if (!is.null(limite_inf) && (length(limite_inf) != n_variables)) {
    stop(paste(
      "limite_inf debe tener un valor por cada variable.",
      "Si para alguna variable no se quiere límite, emplear NA.",
```

```
    "Ejemplo: lim_sup = c(10, NA, 10)"
  ))
}

if (!is.null(limite_sup) && length(limite_sup) != n_variables) {
  stop(paste(
    "limite_sup debe tener un valor por cada variable.",
    "Si para alguna variable no se quiere límite, emplear NA.",
    "Ejemplo: lim_sup = c(10, NA, 10)"
  ))
}

if (is.null(limite_sup) | is.null(limite_inf)) {
  warning(paste(
    "Es altamente recomendable indicar los límites dentro de los",
    "cuales debe buscarse la solución de cada variable.",
    "Por defecto se emplea [-10^3, 10^3]."
  ))
}

if (any(any(is.na(limite_sup)), any(is.na(limite_inf)))) {
  warning(paste(
    "Los límites empleados por defecto cuando no se han definido son:",
    " [-10^3, 10^3]."
  ))
  cat("\n")
}

# Si no se especifica limite_inf, el valor mínimo que pueden tomar las variables
# es -10^3.
if (is.null(limite_inf)) {
  limite_inf <- rep(x = -10^3, times = n_variables)
}

# Si no se especifica limite_sup, el valor máximo que pueden tomar las variables
# es 10^3.
if (is.null(limite_sup)) {
  limite_sup <- rep(x = 10^3, times = n_variables)
}

# Si los límites no son nulos, se reemplazan aquellas posiciones NA por el valor
# por defecto -10^3 y 10^3
if (!is.null(limite_inf)) {
  limite_inf[is.na(limite_inf)] <- -10^3
}

if (!is.null(limite_sup)) {
  limite_sup[is.na(limite_sup)] <- 10^3
}
```

```
  # Matriz donde almacenar los individuos generados.
  poblacion <- matrix(data = NA, nrow = n_poblacion, ncol = n_variables)

  # Bucle para crear cada individuo.
  for (i in 1:n_poblacion) {
    # Se crea un vector de NA que representa el individuo.
    individuo <- rep(NA, times = n_variables)

    for (j in 1:n_variables) {
      # Para cada posición, se genera un valor aleatorio dentro del rango permitido
      # para cada variable.
      individuo[j] <- runif(n = 1, min = limite_inf[j], max = limite_sup[j])
    }
    # Se añade el nuevo individuo a la población.
    poblacion[i, ] <- individuo
  }

  if (verbose) {
    print("Población inicial creada")
    print(paste("Número de individuos =", n_poblacion))
    print(paste(
      "Límites inferiores de cada variable:",
      paste(limite_inf, collapse = ", ")
    ))
    print(paste(
      "Límites superiores de cada variable:",
      paste(limite_sup, collapse = ", ")
    ))
    cat("\n")
  }

  return(poblacion)
}
```

Ejemplo

Se crea una población de 10 individuos de longitud 2, con los valores de la primera variable acotados entre [-100, +100] y la segunda con únicamente el límite inferior [-20, NA].

```
poblacion <- crear_poblacion(
  n_poblacion = 10,
  n_variables = 2,
  limite_inf  = c(-100, -20),
  limite_sup  = c(+100, NA),
  verbose     = TRUE
)
```

```
## [1] "Población inicial creada"
## [1] "Número de individuos = 10"
## [1] "Límites inferiores de cada variable: -100, -20"
## [1] "Límites superiores de cada variable: 100, 1000"
```

```
poblacion
```

```
##            [,1]     [,2]
##  [1,]  99.85059 729.5950
##  [2,]  52.40077 791.5250
##  [3,]  32.85585 864.8343
##  [4,]  84.08724 806.6890
##  [5,] -57.74912 841.8518
##  [6,]  55.44447 626.9558
##  [7,] -98.30764 253.3776
##  [8,]  78.48753 302.8188
##  [9,] -58.75075 232.1449
## [10,]  83.81986 620.1638
```

Fitness de un individuo

Cada individuo de la población debe ser evaluado para cuantificar cómo de bueno es como solución al problema, a esta cuantificación se le llama (*fitness*). Dependiendo de si se trata de un problema de maximización o minimización, la relación del *fitness* con la función objetivo f puede ser:

- Maximización: el individuo tiene mayor *fitness* cuanto mayor es el valor de la función objetivo $f(individuo)$.
- Minimización: el individuo tiene mayor *fitness* cuanto menor es el valor de la función objetivo $f(individuo)$, o lo que es lo mismo, cuanto mayor es el valor de la función objetivo, menor el *fitness*. Tal y como se describe más adelante, el algoritmo genético selecciona los individuos de mayor *fitness*, por lo que, para problemas de minimización, el *fitness* puede calcularse como $1 - f(individuo)$ o también $\frac{1}{1+f(individuo)}$.

```r
calcular_fitness_individuo <- function(individuo, funcion_objetivo, optimizacion,
                                       verbose = TRUE, ...) {
  # Esta función devuelve el fitness de cada individuo de una población.
  #
  # ARGUMENTOS
  # ============================================================================
  # individuo:        vector con los valores de cada variable. El orden de los
  #                   valores debe coincidir con el de los argumentos de la
  #                   función.
  # funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
  #                   sido definida previamente.
  # optimizacion:    "maximizar" o "minimizar". Dependiendo de esto, la relación
  #                   del fitness es directamente o indirectamente proporcional
  #                   al valor de la función.
  # verbose:          mostrar información del proceso por pantalla.
  #
  # RETORNO
  # ============================================================================
  # fitness del individuo.

  # Comprobaciones.
  if(length(individuo) != length(names(formals(funcion_objetivo)))){
    stop(paste("Los individuos deben tener tantos valores como argumentos tiene",
               "la función objetivo."))
  }

  # Cálculo fitness.
  if (optimizacion == "maximizar") {
    fitness <- do.call(funcion_objetivo, args = as.list(individuo))
  } else if (optimizacion == "minimizar") {
    fitness <- 1 - do.call(funcion_objetivo, args = as.list(individuo))
  } else {
    stop("El argumento optimización debe ser maximizar o minimizar.")
  }

  if (verbose) {
    print(paste("El fitness calculado para", optimizacion, "es de:", fitness))
    cat("\n")
  }
  return(fitness)
}
```

Ejemplo

Se calcula el *fitness* del individuo $(x_1 = 10, x_2 = 10)$ para los casos de maximización y minimización de la función $f(x_1, x_2) = x_1 + x_2$.

```
# Función objetivo a optimizar.
funcion <- function(x1, x2) {
  return(x1 + x2)
}

calcular_fitness_individuo(
  individuo = c(10, 10),
  funcion_objetivo = funcion,
  optimizacion = "maximizar",
  verbose = TRUE
)
```

```
## [1] "El fitness calculado para maximizar es de: 20"
```

```
## [1] 20
```

```
calcular_fitness_individuo(
  individuo = c(10, 10),
  funcion_objetivo = funcion,
  optimizacion = "minimizar",
  verbose = TRUE
)
```

```
## [1] "El fitness calculado para minimizar es de: -19"
```

```
## [1] -19
```

Fitness de todos los individuos de una población

Esta función recibe como argumentos una población de individuos, una función objetivo y el tipo de optimización, y devuelve el *fitness* de todos los individuos.

```
calcular_fitness_poblacion <- function(poblacion, funcion_objetivo, optimizacion,
                                       verbose = TRUE, ...) {
  # Esta función devuelve el fitness de cada individuo de una población.
  #
  # ARGUMENTOS
  # ============================================================================
  # poblacion:        matriz que representa la población de individuos.
  # funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
  #                   sido definida previamente.
  # optimizacion:     "maximizar" o "minimizar". Dependiendo de esto, la relación
  #                   del fitness es directamente o indirectamente proporcional
  #                   al valor de la función.
  # verbose:          mostrar información del proceso por pantalla.
  #
  # RETORNO
  # ============================================================================
  # Vector con el fitness de todos los individuos de la población. El orden de
  # los valores se corresponde con el orden de las filas de la matriz población.

  # Vector donde almacenar el fitness de cada individuo.
  fitness_poblacion <- rep(NA, times = nrow(poblacion))

  for (i in 1:nrow(poblacion)) {
    individuo <- poblacion[i, ]

    if (verbose) {
      print(paste("Individuo", i, ":", paste(individuo, collapse = " ")))
    }

    fitness_individuo <- calcular_fitness_individuo(
      individuo = individuo,
      funcion_objetivo = funcion_objetivo,
      optimizacion = optimizacion,
      verbose = verbose
    )
    fitness_poblacion[i] <- fitness_individuo
  }

  if (verbose) {
    print(paste(
      "Fitness calculado para los",
      nrow(poblacion),
```

```
      "individuos de la población."
    ))
    cat("\n")
  }
  return(fitness_poblacion)
}
```

Ejemplo

Se calcula el *fitness* de todos los individuos de una población formada por 5 individuos.

```
# Función objetivo a optimizar.
funcion <- function(x1, x2) {
  return(x1 + x2)
}

# Población simulada.
poblacion <- crear_poblacion(
  n_poblacion = 5,
  n_variables = 2,
  limite_inf = c(-10, -10),
  limite_sup = c(+10, +10),
  verbose = TRUE
)
```

```
## [1] "Población inicial creada"
## [1] "Número de individuos = 5"
## [1] "Límites inferiores de cada variable: -10, -10"
## [1] "Límites superiores de cada variable: 10, 10"
```

```
# Cálculo del fitness de todos los individuos.
fitness_poblacion <- calcular_fitness_poblacion(
  poblacion = poblacion,
  funcion_objetivo = funcion,
  optimizacion = "minimizar",
  verbose = TRUE
)
```

```
## [1] "Individuo 1 : 8.55078153777868 5.6125360308215"
## [1] "El fitness calculado para minimizar es de: -13.1633175686002"
##
## [1] "Individuo 2 : -5.39868548512459 -4.43999414332211"
## [1] "El fitness calculado para minimizar es de: 10.8386796284467"
##
## [1] "Individuo 3 : 5.08252339437604 2.90823311079293"
## [1] "El fitness calculado para minimizar es de: -6.99075650516897"
##
## [1] "Individuo 4 : 4.87361517269164 8.33372023422271"
```

```
## [1] "El fitness calculado para minimizar es de: -12.2073354069144"
##
## [1] "Individuo 5 : 9.16035243310034 4.57680635154247"
## [1] "El fitness calculado para minimizar es de: -12.7371587846428"
##
## [1] "Fitness calculado para los 5 individuos de la población."
```

```
fitness_poblacion
```

```
## [1] -13.163318  10.838680  -6.990757 -12.207335 -12.737159
```

El vector devuelto contiene el *fitness* de cada uno de los individuos en el mismo orden que se encuentran en la matriz de la población.

Seleccionar individuos

La forma en que se seleccionan los individuos que participan en cada cruce difiere en las distintas implementaciones de los algoritmos genéticos. Por lo general, todas ellas tienden a favorecer la selección de aquellos individuos con mayor *fitness*. Algunas de las estrategias más comunes son:

- Método de ruleta: la probabilidad de que un individuo sea seleccionado es proporcional a su *fitness* relativo, es decir, a su *fitness* dividido por la suma del *fitness* de todos los individuos de la población. Si el *fitness* de un individuo es el doble que el de otro, también lo será la probabilidad de que sea seleccionado. Este método presenta problemas si el *fitness* de unos pocos individuos es muy superior (varios órdenes de magnitud) al resto, ya que estos serán seleccionados de forma repetida y casi todos los individuos de la siguiente generación serán "hijos" de los mismos "padres" (poca variación).
- Método *rank*: la probabilidad de selección de un individuo es inversamente proporcional a la posición que ocupa tras ordenar todos los individuos de mayor a menor *fitness*. Este método es menos agresivo que el método ruleta cuando la diferencia entre los mayores *fitness* es varios órdenes de magnitud superior al resto.
- Selección competitiva (*tournament*): se seleccionan aleatoriamente dos parejas de individuos de la población (todos con la misma probabilidad). De cada pareja se selecciona el que tenga mayor *fitness*. Finalmente, se comparan los dos finalistas y se selecciona el de mayor *fitness*. Este método tiende a generar una distribución de la probabilidad de selección más equilibrada que las dos anteriores.
- Selección truncada (*truncated selection*): se realizan selecciones aleatorias de individuos, habiendo descartado primero los *n* individuos con menor *fitness* de la población.

```r
seleccionar_individuo <- function(vector_fitness, metodo_seleccion = "tournament",
                                  verbose = FALSE) {
  # Esta función recibe como argumento un vector con el fitness de cada individuo
  # y selecciona una de las posiciones, donde la probabilidad de selección es
  # proporcional al fitness.

  # ARGUMENTOS
  # ============================================================================
  # vector_fitness:   un vector con el fitness de cada individuo.
  # metodo_seleccion: método para establecer la probabilidad de selección. Puede
  #                   ser: "ruleta", "rank", o "tournament".
  # verbose:          mostrar información del proceso por pantalla.
  #
  # RETORNO
  # ============================================================================
  # El índice que ocupa el individuo seleccionado.

  if (metodo_seleccion == "ruleta") {
    probabilidad_seleccion <- (vector_fitness) / sum(vector_fitness)

    ind_seleccionado <- sample(
      x = 1:length(vector_fitness),
      size = 1,
      prob = probabilidad_seleccion
    )
  } else if (metodo_seleccion == "rank") {
    probabilidad_seleccion <- 1 / rank(-vector_fitness)

    ind_seleccionado <- sample(
      x = 1:length(vector_fitness),
      size = 1,
      prob = probabilidad_seleccion
    )
  } else if (metodo_seleccion == "tournament") {
    # Se seleccionan aleatoriamente dos parejas de individuos.
    ind_candidatos_a <- sample(x = 1:length(vector_fitness), size = 2)
    ind_candidatos_b <- sample(x = 1:length(vector_fitness), size = 2)

    # De cada pareja se selecciona el de mayor fitness.
    ind_ganador_a <- ifelse(
      vector_fitness[ind_candidatos_a[1]] > vector_fitness[ind_candidatos_a[2]],
      ind_candidatos_a[1],
      ind_candidatos_a[2]
    )
    ind_ganador_b <- ifelse(
      vector_fitness[ind_candidatos_b[1]] > vector_fitness[ind_candidatos_b[2]],
      ind_candidatos_b[1],
      ind_candidatos_b[2]
    )
```

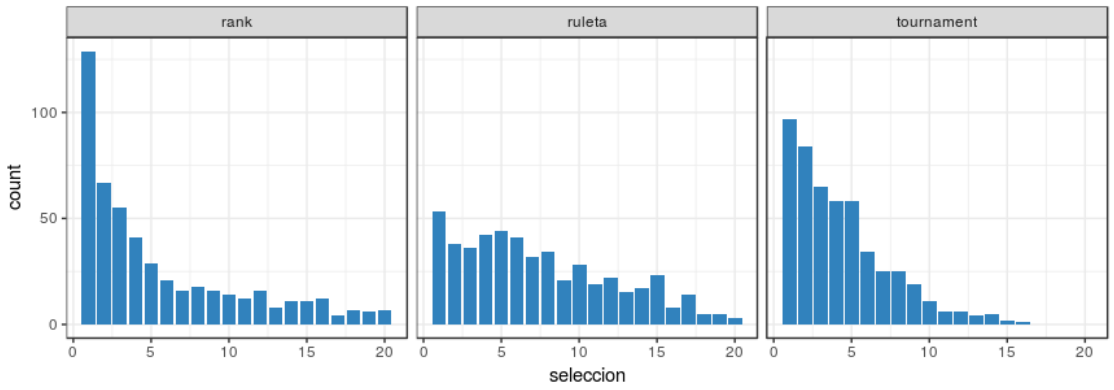
```
    # Se comparan los dos ganadores de cada pareja.
    ind_seleccionado <- ifelse(
      vector_fitness[ind_ganador_a] > vector_fitness[ind_ganador_b],
      ind_ganador_a,
      ind_ganador_b
    )
  } else {
    stop("El argumento metodo_seleccion debe ser: ruleta, rank o tournament")
  }

  if (verbose) {
    print(paste("Método de selección empleado:", metodo_seleccion))
  }

  return(ind_seleccionado)
}
```

Ejemplo

Se compara el patrón de selección del mejor individuo entre los métodos ruleta, *rank* y *tournament*. En primer lugar, se muestra un caso en el que la diferencia entre el mayor y el menor de los *fitness* no es muy acusada, y un segundo caso en el que sí lo es.

```
library(tidyverse)

fitness_poblacion <- c(20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5,
                       4, 3, 2, 1)

selecciones_ruleta <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_ruleta[i] <- seleccionar_individuo(
    vector_fitness   = fitness_poblacion,
    metodo_seleccion = "ruleta"
  )
}
selecciones_ruleta <- data.frame(seleccion = selecciones_ruleta) %>%
                      mutate(metodo_seleccion = "ruleta")

selecciones_rank <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_rank[i] <- seleccionar_individuo(
    vector_fitness   = fitness_poblacion,
    metodo_seleccion = "rank"
  )
```

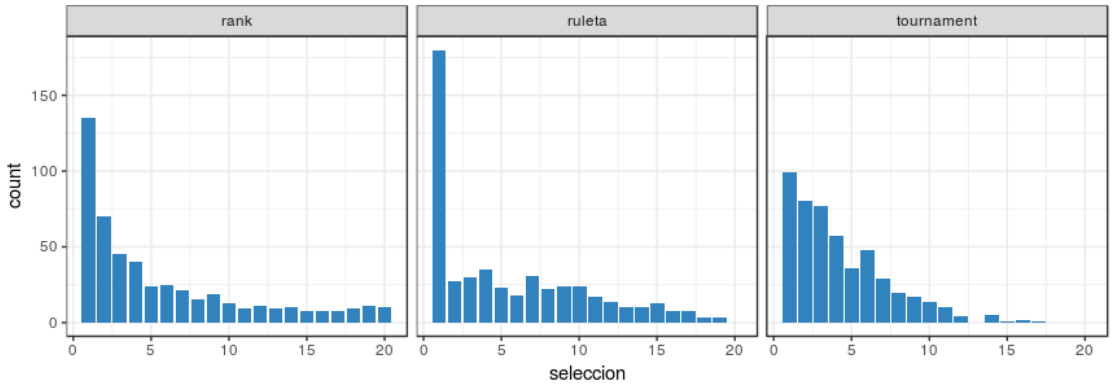
```
}
selecciones_rank <- data.frame(seleccion = selecciones_rank) %>%
                    mutate(metodo_seleccion = "rank")

selecciones_tournament <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_tournament[i] <- seleccionar_individuo(
    vector_fitness   = fitness_poblacion,
    metodo_seleccion = "tournament"
  )
}
selecciones_tournament <- data.frame(seleccion = selecciones_tournament) %>%
  mutate(metodo_seleccion = "tournament")

bind_rows(selecciones_ruleta, selecciones_rank, selecciones_tournament) %>%
  ggplot(aes(x= seleccion)) +
  geom_bar(fill = "#3182bd") +
  facet_grid(. ~ metodo_seleccion) +
  theme_bw()
```



Cuando no existe una gran diferencia entre el individuo de mayor *fitness* y el resto, con el método *rank*, el individuo con mayor *fitness* se selecciona con mucha más frecuencia que el resto. Con los otros dos métodos, la probabilidad de selección decae de forma más gradual.

```
fitness_poblacion <- c(100, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5,
                       4, 3, 2, 1)

selecciones_ruleta <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_ruleta[i] <- seleccionar_individuo(
    vector_fitness = fitness_poblacion,
    metodo_seleccion = "ruleta"
  )
}
selecciones_ruleta <- data.frame(seleccion = selecciones_ruleta) %>%
```

```
                      mutate(metodo_seleccion = "ruleta")

selecciones_rank <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_rank[i] <- seleccionar_individuo(
    vector_fitness = fitness_poblacion,
    metodo_seleccion = "rank"
  )
}
selecciones_rank <- data.frame(seleccion = selecciones_rank) %>%
                    mutate(metodo_seleccion = "rank")


selecciones_tournament <- rep(NA, times = 500)
for (i in 1:500) {
  selecciones_tournament[i] <- seleccionar_individuo(
    vector_fitness = fitness_poblacion,
    metodo_seleccion = "tournament"
  )
}
selecciones_tournament <- data.frame(seleccion = selecciones_tournament) %>%
                          mutate(metodo_seleccion = "tournament")

bind_rows(selecciones_ruleta, selecciones_rank, selecciones_tournament) %>%
  ggplot(aes(x = seleccion)) +
  geom_bar(fill = "#3182bd") +
  facet_grid(. ~ metodo_seleccion) +
  theme_bw()
```



Cuando existe una gran diferencia entre el individuo de mayor *fitness* y el resto (uno o varios órdenes de magnitud), con el método *ruleta*, el individuo con mayor *fitness* se selecciona con mucha más frecuencia que el resto. A diferencia del caso anterior, en esta situación, la probabilidad de selección decae de forma más gradual con los métodos *rank* y *tournament*.

Teniendo en cuenta los comportamientos de selección de cada método, el método *tournament* parece ser la opción más equilibrada.

Cruzar dos individuos (*crossover*, recombinación)

El objetivo de esta etapa es generar, a partir de individuos ya existentes (parentales), nuevos individuos (descendencia) que combinen las características de los anteriores. Este es otro de los puntos del algoritmo en los que se puede seguir varias estrategias. Tres de las más empleadas son:

- Cruzamiento a partir de uno solo punto: se selecciona aleatoriamente una posición que actúa como punto de corte. Cada individuo parental se divide en dos partes y se intercambian las mitades. Como resultado de este proceso, por cada cruce, se generan dos nuevos individuos.
- Cruzamiento a partir múltiples puntos: se seleccionan aleatoriamente varias posiciones que actúan como puntos de corte. Cada individuo parental se divide por los puntos de corte y se intercambian las partes. Como resultado de este proceso, por cada cruce, se generan dos nuevos individuos.
- Cruzamiento uniforme: el valor que toma cada posición del nuevo individuo se obtiene de uno de los dos parentales. Por lo general, la probabilidad de que el valor proceda de cada parental es la misma, aunque podría, por ejemplo, estar condicionada al *fitness* de cada uno. A diferencia de las anteriores estrategias, con esta, de cada cruce se genera un único descendiente.

```
cruzar_individuos <- function(parental_1, parental_2) {
  # Esta función devuelve un individuo resultado de cruzar dos individuos
  # parentales con el método de cruzamiento uniforme.
  #
  # ARGUMENTOS
  # ============================================================================
  # parental_1: vector que representa a un individuo.
  # parental_2: vector que representa a un individuo.

  # RETORNO
  # ============================================================================
  # Un vector que representa a un nuevo individuo.

  if (length(parental_1) != length(parental_2)) {
    stop(paste0(
      "La longitud de los dos vectores que representan a los ",
      "individuos debe ser la misma."
    ))
  }

  # Se crea el vector que representa el nuevo individuo
  descendencia <- rep(NA, times = length(parental_1))
```

```
  # Se seleccionan aleatoriamente las posiciones que se heredan del parental_1.
  herencia_parent_1 <- sample(
    x = c(TRUE, FALSE),
    size = length(parental_1),
    replace = TRUE
  )
  # El resto de posiciones se heredan del parental_2.
  herencia_parent_2 <- !(herencia_parent_1)

  descendencia[herencia_parent_1] <- parental_1[herencia_parent_1]
  descendencia[herencia_parent_2] <- parental_2[herencia_parent_2]

  return(descendencia)
}
```

Ejemplo

Se obtiene un nuevo individuo a partir del cruce de los individuos `c(T, T, T, T, T)` y `c(F, F, F, F, F)`.

```
cruzar_individuos(parental_1 = c(T, T, T, T, T),
                  parental_2 =  c(F, F, F, F, F))
```

```
## [1]  TRUE FALSE FALSE  TRUE  TRUE
```

Mutar individuo

Tras generar cada nuevo individuo de la descendencia, este se somete a un proceso de mutación en el que, cada una de sus posiciones, puede verse modificada con una probabilidad p. Este paso es importante para añadir diversidad al proceso y evitar que el algoritmo caiga en mínimos locales por que todos los individuos sean demasiado parecidos de una generación a otra.

Existen diferentes estrategias para controlar la magnitud del cambio que puede provocar una mutación.

- Distribución uniforme: la mutación de la posición i se consigue sumándole al valor de i un valor extraído de una distribución uniforme, por ejemplo una entre [-1,+1].
- Distribución normal: la mutación de la posición i se consigue sumándole al valor de i un valor extraído de una distribución normal, comúnmente centrada en 0 y con una

determinada desviación estándar. Cuanto mayor la desviación estándar, con mayor probabilidad la mutación introducirá cambios grandes.

- Aleatorio: la mutación de la posición *i* se consigue reemplazando el valor de *i* por nuevo valor aleatorio dentro del rango permitido para esa variable. Esta estrategia suele conllevar mayores variaciones que las dos anteriores.

Hay que tener en cuenta que, debido a las mutaciones, un valor que inicialmente estaba dentro del rango permitido puede salirse de él. Una forma de evitarlo es: si el valor tras la mutación excede alguno de los límites acotados, se sobrescribe con el valor del límite. Es decir, se permite que los valores se alejen como máximo hasta el límite impuesto.

```r
mutar_individuo <- function(individuo, limite_inf, limite_sup,
                            prob_mut = 0.01, distribucion = "uniforme",
                            media_distribucion = 1, sd_distribucion = 1,
                            min_distribucion = -1, max_distribucion = 1) {
  # ARGUMENTOS
  # ============================================================================
  # individuo: vector que representa a un individuo.
  # prob_mut:  probabilidad que tiene cada posición del individuo de mutar.
  # distribucion: distribución de la que obtener el factor de mutación. Puede
  #               ser: "normal", "uniforme" o "aleatoria".
  # media_distribucion: media de la distribución si se selecciona
  #                     distribucion = "normal".
  # sd_distribucion:    desviación estándar de la distribución si se selecciona
  #                     distribucion = "normal".
  # min_distribucion:   mínimo la distribución si se selecciona
  #                     distribucion = "uniforme".
  # max_distribucion:   máximo la distribución si se selecciona
  #                     distribucion = "uniforme".
  #
  # RETORNO
  # ============================================================================
  # Un vector que representa al individuo tras someterse a las mutaciones.

  # Selección de posiciones a mutar.
  posiciones_mutadas <- runif(n = length(individuo), min = 0, max = 1) < prob_mut

  # Se modifica el valor de aquellas posiciones que hayan sido seleccionadas para
  # mutar. Si el valor de prob_mut es muy bajo, las mutaciones serán muy poco
  # frecuentes y el individuo devuelto será casi siempre igual al original.

  # Si se emplea distribucion = "uniforme" o distribucion = "normal":
  if (distribucion == "normal" | distribucion == "uniforme") {
    # Se extrae un valor aleatorio de la distribución elegida que se suma
    # para modificar la/las posiciones mutadas.
```

```
    if (distribucion == "normal") {
      factor_mut <- rnorm(
        n = sum(posiciones_mutadas),
        mean = media_distribucion,
        sd = sd_distribucion
      )
    }
    if (distribucion == "uniforme") {
      factor_mut <- runif(
        n = sum(posiciones_mutadas),
        min = min_distribucion,
        max = max_distribucion
      )
    }

    individuo[posiciones_mutadas] <- individuo[posiciones_mutadas] + factor_mut

    # Se comprueba si algún valor mutado supera los límites impuestos. En tal caso
    #  se sobrescribe con el valor del límite correspondiente.
    for (i in which(posiciones_mutadas)) {
      if (individuo[i] < limite_inf[i]) {
        individuo[i] <- limite_inf[i]
      }
      if (individuo[i] > limite_sup[i]) {
        individuo[i] <- limite_sup[i]
      }
    }
  } else if (distribucion == "aleatoria") {
    for (i in which(posiciones_mutadas)) {
      individuo[i] <- runif(n = 1, min = limite_inf[i], max = limite_sup[i])
    }
  } else {
    stop(paste("El argumento distribución debe ser: normal, uniforme o aleatoria.")
)
  }

  return(individuo)
}
```

Ejemplo

Se somete a un individuo al proceso de mutación, con una probabilidad de mutación de 0.5 y con límites para el valor que puede tener cada posición.

```
mutar_individuo(
  individuo = c(3, 3, 3, 3, 3, 3),
  prob_mut = 0.5,
  distribucion = "aleatoria",
  media_distribucion = 1,
  sd_distribucion = 1,
  limite_inf = c(-5, -5, -5, -5, -5, -5),
  limite_sup = c(5, 5, 5, 5, 5, 5)
)
```

```
## [1]  1.0770041  3.0000000 -0.1646922  3.0000000 -1.7318666  3.0000000
```

Algoritmo completo

En cada uno de los apartados anteriores se ha definido una de las etapas del algoritmo genético. A continuación, se combinan todas ellas dentro de una única función.

```
optimizar_ga <- function(
                         funcion_objetivo,
                         n_variables,
                         optimizacion,
                         limite_inf = NULL,
                         limite_sup = NULL,
                         n_poblacion = 20,
                         n_generaciones = 50,
                         elitismo = 0.1,
                         prob_mut = 0.01,
                         distribucion = "uniforme",
                         media_distribucion = 1,
                         sd_distribucion = 1,
                         min_distribucion = -1,
                         max_distribucion = 1,
                         metodo_seleccion = "tournament",
                         parada_temprana = FALSE,
                         rondas_parada = NULL,
                         tolerancia_parada = NULL,
                         verbose = FALSE,
                         ...) {
```

```
# ARGUMENTOS
# ============================================================================
# funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
#                   sido definida previamente.
# n_variables:      longitud de los individuos.
# optimizacion:     "maximizar" o "minimizar". Dependiendo de esto, la relación
#                   del fitness es directamente o indirectamente proporcional al
#                   valor de la función.
# limite_inf:       vector con el límite inferior de cada variable. Si solo se
#                   quiere imponer límites a algunas variables, emplear NA para
#                   las que no se quiere acotar.
# limite_sup:       vector con el límite superior de cada variable. Si solo se
#                   quiere imponer límites a algunas variables, emplear NA para
#                   las que no se quieren acotar.
# n_poblacion:      número total de individuos de la población.
# n_generaciones:   número total de generaciones creadas.
# elitismo:         porcentaje de mejores individuos de la población actual que
#                   pasan directamente a la siguiente población.
# prob_mut:         probabilidad que tiene cada posición del individuo de mutar.
# distribucion:     distribución de la que obtener el factor de mutación. Puede
#                   ser: "normal", "uniforme" o "aleatoria".
# media_distribucion: media de la distribución si se selecciona
#                     distribucion="normal".
# sd_distribucion:  desviación estándar de la distribución si se selecciona
#                   distribucion="normal".
# min_distribucion: mínimo la distribución si se selecciona
#                   distribucion="uniforme".
# max_distribucion: máximo la distribución si se selecciona
#                   distribucion="uniforme".
# metodo_seleccion: método para establecer la probabilidad de selección. Puede
#                   ser: "ruleta", "rank" o "tournament".
# parada_temprana:  si durante las últimas "rondas_parada" generaciones la
#                   diferencia absoluta entre mejores individuos no es superior
#                   al valor de "tolerancia_parada", se detiene el algoritmo y no
#                   se crean nuevas generaciones.
# rondas_parada:    número de generaciones consecutivas sin mejora mínima para
#                   que se active la parada temprana.
# tolerancia_parada: valor mínimo que debe tener la diferencia de generaciones
#                    consecutivas para considerar que hay cambio.
# verbose:          TRUE para que se imprima por pantalla el resultado de cada
#                   paso del algoritmo.
#
# RETORNO
# ============================================================================
# La función devuelve una lista con 5 elementos:
# fitness:            una lista con el fitness del mejor individuo de cada
#                     generación.
# mejores_individuos: una lista con la combinación de predictores del mejor
#                     individuo de cada generación.
# mejor_individuo:    combinación de predictores del mejor individuo encontrado
```

```
#                     en todo el proceso.
# diferencia_abs:     una lista con la diferencia absoluta entre el fitness
#                     del mejor individuo de generaciones consecutivas.
# df_resultados:      un dataframe con todos los resultados anteriores.

# COMPROBACIONES INICIALES
# ==============================================================================
# Si se activa la parada temprana, hay que especificar los argumentos
# rondas_parada y tolerancia_parada.
if (isTRUE(parada_temprana) &&
    (is.null(rondas_parada) | is.null(tolerancia_parada)) ) {
  stop(paste(
    "Para activar la parada temprana es necesario indicar un valor",
    "de rondas_parada y de tolerancia_parada."
  ))
}

# ESTABLECER LOS LÍMITES DE BÚSQUEDA SI EL USUARIO NO LO HA HECHO
# ==============================================================================
if (is.null(limite_sup) | is.null(limite_inf)) {
  warning(paste(
    "Es altamente recomendable indicar los límites dentro de los",
    "cuales debe buscarse la solución de cada variable.",
    "Por defecto se emplea: [-10^3, 10^3]."
  ))
}

if (any(
      is.null(limite_sup), is.null(limite_inf), any(is.na(limite_sup)),
      any(is.na(limite_inf))
)) {
  warning(paste(
    "Los límites empleados por defecto cuando no se han definido son:",
    " [-10^3, 10^3]."
  ))
  cat("\n")
}

# Si no se especifica limite_inf, el valor mínimo que pueden tomar las variables
# es -10^3.
if (is.null(limite_inf)) {
  limite_inf <- rep(x = -10^3, times = n_variables)
}

# Si no se especifica limite_sup, el valor máximo que pueden tomar las variables
# es 10^3.
if (is.null(limite_sup)) {
  limite_sup <- rep(x = 10^3, times = n_variables)
}
```

```r
# Si los límites no son nulos, se reemplazan aquellas posiciones NA por el valor
# por defecto -10^3 y 10^3.
if (!is.null(limite_inf)) {
  limite_inf[is.na(limite_inf)] <- -10^3
}

if (!is.null(limite_sup)) {
  limite_sup[is.na(limite_sup)] <- 10^3
}

# ALMACENAMIENTO DE RESULTADOS
# ==============================================================================
# Por cada generación se almacena el mejor individuo, su fitness, y la diferencia
# absoluta respecto a la última generación.
resultados_fitness <- vector(mode = "list", length = n_generaciones)
resultados_individuo <- vector(mode = "list", length = n_generaciones)
diferencia_abs <- vector(mode = "list", length = n_generaciones)

# CREACIÓN DE LA POBLACIÓN INICIAL
# ==============================================================================
poblacion <- crear_poblacion(
  n_poblacion = n_poblacion,
  n_variables = n_variables,
  limite_inf  = limite_inf,
  limite_sup  = limite_sup,
  verbose     = verbose
)
# ITERACIÓN DE POBLACIONES
# ==============================================================================
for (i in 1:n_generaciones) {
  if (verbose) {
    print("----------------------")
    print(paste("Generación:", i))
    print("----------------------")
  }

  # CALCULAR FITNESS DE LOS INDIVIDUOS DE LA POBLACIÓN
  # ============================================================================
  fitness_ind_poblacion <- calcular_fitness_poblacion(
    poblacion    = poblacion,
    funcion_objetivo = funcion_objetivo,
    optimizacion = optimizacion,
    verbose      = verbose
  )

  # SE ALMACENA EL MEJOR INDIVIDUO DE LA POBLACIÓN ACTUAL
  # ============================================================================
  fitness_mejor_individuo <- max(fitness_ind_poblacion)
```

```
  mejor_individuo <- poblacion[which.max(fitness_ind_poblacion), ]
  resultados_fitness[[i]]   <- fitness_mejor_individuo
  resultados_individuo[[i]] <- mejor_individuo

  # SE CALCULA LA DIFERENCIA ABSOLUTA RESPECTO A LA GENERACIÓN ANTERIOR
  # ============================================================================
  # La diferencia solo puede calcularse a partir de la segunda generación.
  if (i > 1) {
    diferencia_abs[[i]]<-abs(resultados_fitness[[i - 1]]-resultados_fitness[[i]])
  }

  # NUEVA POBLACIÓN
  # ============================================================================
  nueva_poblacion <- matrix(
    data = NA,
    nrow = nrow(poblacion),
    ncol = ncol(poblacion)
  )

  # ELITISMO
  # ============================================================================
  # El elitismo indica el porcentaje de mejores individuos de la población
  # actual que pasan directamente a la siguiente población. De esta forma, se
  # asegura que, la siguiente generación, no sea nunca inferior.

  if (elitismo > 0) {
    n_elitismo <- ceiling(nrow(poblacion) * elitismo)
    posicion_n_mejores <- order(fitness_ind_poblacion, decreasing = TRUE)
    posicion_n_mejores <- posicion_n_mejores[1:n_elitismo]
    nueva_poblacion[1:n_elitismo, ] <- poblacion[posicion_n_mejores, ]
  } else {
    n_elitismo <- 0
  }

  # CREACIÓN DE NUEVOS INDIVIDUOS POR CRUCES
  # ============================================================================
  for (j in (n_elitismo + 1):nrow(nueva_poblacion)) {
    # Seleccionar parentales
    indice_parental_1 <- seleccionar_individuo(
      vector_fitness   = fitness_ind_poblacion,
      metodo_seleccion = metodo_seleccion
    )
    indice_parental_2 <- seleccionar_individuo(
      vector_fitness   = fitness_ind_poblacion,
      metodo_seleccion = metodo_seleccion
    )
    parental_1 <- poblacion[indice_parental_1, ]
    parental_2 <- poblacion[indice_parental_2, ]
```

```
    # Cruzar parentales para obtener la descendencia
    descendencia <- cruzar_individuos(
      parental_1 = parental_1,
      parental_2 = parental_2
    )
    # Mutar la descendencia
    descendencia <- mutar_individuo(
      individuo  = descendencia,
      prob_mut   = prob_mut,
      limite_inf = limite_inf,
      limite_sup = limite_sup,
      distribucion = distribucion,
      media_distribucion = media_distribucion,
      sd_distribucion  = sd_distribucion,
      min_distribucion = min_distribucion,
      max_distribucion = max_distribucion
    )

    nueva_poblacion[j, ] <- descendencia
  }
  poblacion <- nueva_poblacion

  # CRITERIO DE PARADA
  # ============================================================================
  # Si durante las últimas n generaciones, la diferencia absoluta entre mejores
  # individuos no es superior al valor de tolerancia_parada, se detiene el
  # algoritmo y no se crean nuevas generaciones.

  if (parada_temprana && (i > rondas_parada)) {
    ultimos_n <- tail(unlist(diferencia_abs), n = rondas_parada)
    if (all(ultimos_n < tolerancia_parada)) {
      print(paste(
        "Algoritmo detenido en la generacion", i,
        "por falta cambio mínimo de", tolerancia_parada,
        "durante", rondas_parada,
        "generaciones consecutivas."
      ))
      break()
    }
  }
}

# IDENTIFICACIÓN DEL MEJOR INDIVIDUO DE TODO EL PROCESO
# ============================================================================
mejor_individuo <- resultados_individuo[[which.max(unlist(resultados_fitness))]]
```

```r
  # RESULTADOS
  # ============================================================================
  # Para crear el dataframe se convierten las listas a vectores del mismo tamaño.
  fitness <- unlist(resultados_fitness)
  predictores <- resultados_individuo[!sapply(resultados_individuo, is.null)]
  predictores <- sapply(predictores, function(x) {
    paste(x, collapse = ", ")
  })
  diferencia_abs <- c(NA, unlist(diferencia_abs))

  df_resultados <- data.frame(
    generacion = seq_along(fitness),
    fitness = fitness,
    predictores = predictores,
    diferencia_abs = diferencia_abs
  )

  return(list(
    fitness = resultados_fitness,
    mejores_individuos = resultados_individuo,
    diferencia_abs = diferencia_abs,
    df_resultados = df_resultados,
    mejor_individuo = mejor_individuo
  ))
}
```

Ejemplo 1

En este ejemplo se pretende evaluar la capacidad del algoritmo genético para encontrar el mínimo de la función $f(x_1, x_2) = x_1^2 + x_2^2$. El mínimo global de esta función puede obtenerse de forma analítica igualando las derivadas parciales a cero, lo que permite comparar el resultado obtenido.

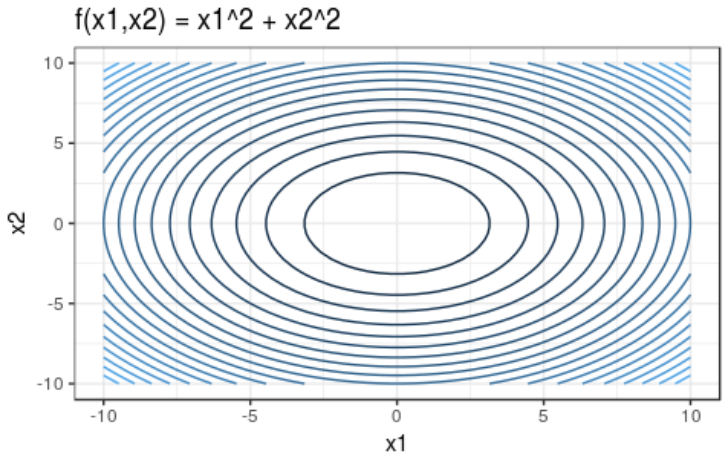
$$f(x_1 = 0, x_2 = 0) = 0$$

Función objetivo

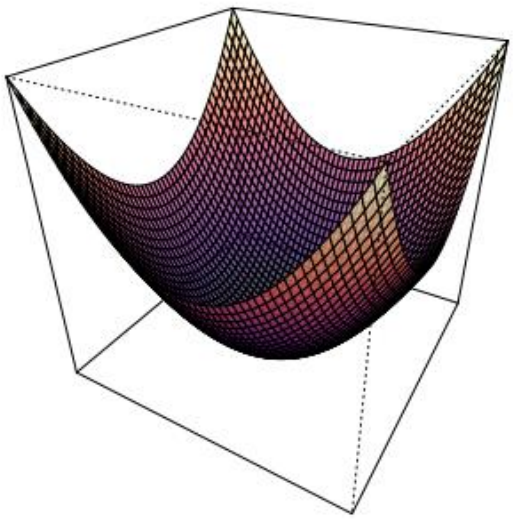
```
# Función objetivo a optimizar.
funcion <- function(x1, x2){
  return(x1^2 + x2^2)
}
```

Representación gráfica de la función.

```
x1 <- seq(-10, 10, length.out = 50)
x2 <- seq(-10, 10, length.out = 50)
datos <- expand.grid(x1 = x1, x2 = x2)
datos <- datos %>%
         mutate(f_x = map2_dbl(x1, x2, .f = funcion))
ggplot(data = datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  labs(title = "f(x1,x2) = x1^2 + x2^2") +
  theme_bw() + theme(legend.position = "none")
```



```
x1  <- seq(-10, 10, length.out = 50)
x2  <- seq(-10, 10, length.out = 50)
f_x <- outer(x1,x2, FUN = funcion)
library(viridis)
colores        <- viridis::magma(n = 100, alpha = 0.7)
z.facet.center <- (f_x[-1, -1] + f_x[-1, -ncol(f_x)] +
                     f_x[-nrow(f_x), -1] +
                     f_x[-nrow(f_x), -ncol(f_x)])/4
z.facet.range  <- cut(z.facet.center, 100)
par(mai = c(0,0,0,0))
persp(x = x1, y = x2, z = f_x, shade = 0.8, phi = 30,
      theta = 30, col = colores[z.facet.range], axes = FALSE)
```



Optimización

```
# Ejemplo ilustrativo, en un caso real, emplear como mínimo 100 individuos por
# generación.
resultados_ga <- optimizar_ga(
                   funcion_objetivo = funcion,
                   n_variables = 2,
                   optimizacion = "minimizar",
                   limite_inf = c(-10, -10),
                   limite_sup = c(10, 10),
                   n_poblacion = 30,
                   n_generaciones = 500,
                   elitismo = 0.01,
                   prob_mut = 0.1,
                   distribucion = "uniforme",
                   min_distribucion = -1,
```

```
                    max_distribucion = 1,
                    metodo_seleccion = "tournament",
                    parada_temprana = TRUE,
                    rondas_parada = 10,
                    tolerancia_parada = 0.0001,
                    verbose = FALSE
                    )
```

```
## [1] "Algoritmo detenido en la generacion 31 por falta cambio mínimo de 1e-04 dur
ante 10 generaciones consecutivas."
```

Resultados

El objeto devuelto por la función `optimizar_ga` almacena la información (*fitness*, valor de las variables,...) del mejor individuo de cada generación.

```
resultados_ga$df_resultados
```

```
##    generacion     fitness                                predictores
## 1           1 -5.81727412       -2.4988521495834, 0.756975593976676
## 2           2 -5.81727412       -2.4988521495834, 0.756975593976676
## 3           3 -0.19260645     -0.787143188063055, 0.756975593976676
## 4           4 -0.19260645     -0.787143188063055, 0.756975593976676
## 5           5 -0.05432426     -0.787143188063055, 0.659340472426265
## 6           6  0.02626519     -0.734169563278556, 0.659340472426265
## 7           7  0.55929533     0.0772969243116677, 0.659340472426265
## 8           8  0.55929533     0.0772969243116677, 0.659340472426265
## 9           9  0.98542937    0.0772969243116677, 0.0927136279642582
## 10         10  0.98542937    0.0772969243116677, 0.0927136279642582
## 11         11  0.98542937    0.0772969243116677, 0.0927136279642582
## 12         12  0.99117111    0.0152668608352542, 0.0927136279642582
## 13         13  0.99139964  -0.00213159853592515, 0.0927136279642582
## 14         14  0.99881607    0.0152668608352542, 0.0308359791524708
## 15         15  0.99904460  -0.00213159853592515, 0.0308359791524708
## 16         16  0.99904460  -0.00213159853592515, 0.0308359791524708
## 17         17  0.99904460  -0.00213159853592515, 0.0308359791524708
## 18         18  0.99904460  -0.00213159853592515, 0.0308359791524708
## 19         19  0.99904460  -0.00213159853592515, 0.0308359791524708
## 20         20  0.99904460  -0.00213159853592515, 0.0308359791524708
## 21         21  0.99934204 -0.00213159853592515, -0.0255620777606964
## 22         22  0.99934204 -0.00213159853592515, -0.0255620777606964
## 23         23  0.99934204 -0.00213159853592515, -0.0255620777606964
## 24         24  0.99934204 -0.00213159853592515, -0.0255620777606964
## 25         25  0.99934204 -0.00213159853592515, -0.0255620777606964
## 26         26  0.99934204 -0.00213159853592515, -0.0255620777606964
## 27         27  0.99934204 -0.00213159853592515, -0.0255620777606964
## 28         28  0.99934204 -0.00213159853592515, -0.0255620777606964
## 29         29  0.99934204 -0.00213159853592515, -0.0255620777606964
## 30         30  0.99934204 -0.00213159853592515, -0.0255620777606964
```

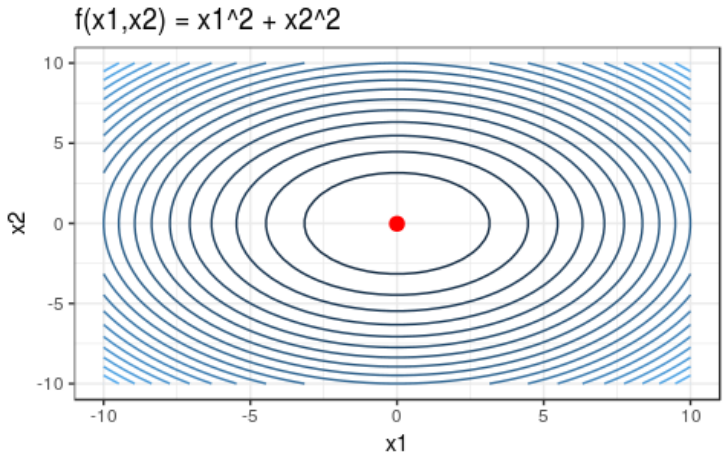
```
## 31         31  0.99934204 -0.00213159853592515, -0.0255620777606964
##    diferencia_abs
## 1              NA
## 2    0.0000000000
## 3    5.6246676670
## 4    0.0000000000
## 5    0.1382821913
## 6    0.0805894509
## 7    0.5330301331
## 8    0.0000000000
## 9    0.4261340418
## 10   0.0000000000
## 11   0.0000000000
## 12   0.0057417375
## 13   0.0002285333
## 14   0.0074164259
## 15   0.0002285333
## 16   0.0000000000
## 17   0.0000000000
## 18   0.0000000000
## 19   0.0000000000
## 20   0.0000000000
## 21   0.0002974378
## 22   0.0000000000
## 23   0.0000000000
## 24   0.0000000000
## 25   0.0000000000
## 26   0.0000000000
## 27   0.0000000000
## 28   0.0000000000
## 29   0.0000000000
## 30   0.0000000000
## 31   0.0000000000
```

Mejor individuo

```
resultados_ga$mejor_individuo
```

```
## [1] -0.002131599 -0.025562078
```

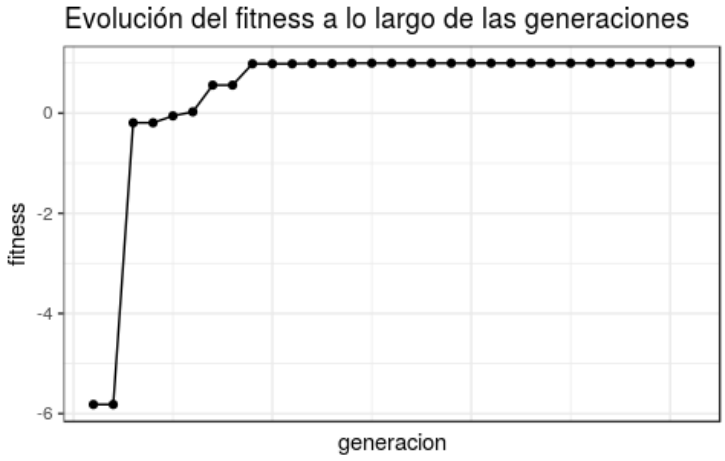
```
ggplot(data = datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  annotate(
    geom = "point",
    x = resultados_ga$mejor_individuo[1],
    y = resultados_ga$mejor_individuo[2],
    color = "red",
    size = 3
  ) +
  labs(title = "f(x1,x2) = x1^2 + x2^2") +
  theme_bw() +  theme(legend.position = "none")
```



Evolución del error

En el siguiente gráfico se puede ver cómo evoluciona el *fitness* del mejor individuo a medida que avanzan las generaciones.

```
library(ggplot2)
ggplot(data = resultados_ga$df_resultados,
       aes(x = generacion, y = fitness)) +
  geom_line(aes(group = 1)) +
  geom_point() +
  labs(title = "Evolución del fitness a lo largo de las generaciones") +
  theme_bw() +
  theme(axis.text.x = element_blank(),
        axis.ticks.x = element_blank())
```



Animación de cómo avanza la búsqueda del mínimo (para versión html)

```
library(gganimate)
evolucion_resultados <- resultados_ga$df_resultados %>%
                        separate(col = predictores, sep = ",",
                                 into = c("x1", "x2")) %>%
                        mutate(x1 = as.numeric(x1),
                               x2 = as.numeric(x2)) %>%
                        select(generacion, x1, x2)

gift <- ggplot() +
        geom_contour(data = datos,
                     aes(x = x1, y = x2, z = f_x, colour = stat(level)),
                     bins = 20) +
        geom_point(data = evolucion_resultados,
                   aes(x = x1, y = x2),
                   color = "red",
                   size = 2.3) +
        theme_bw() +
        theme(legend.position = "none") +
        transition_manual(frames = generacion) +
        ggtitle("Posición del mínimo encontrado",
                subtitle = "Generación {frame}")

animate(plot = gift)
```

Ejemplo 2

En este ejemplo se pretende evaluar la capacidad del algoritmo genético para encontrar el mínimo de la función de *Mishra Bird*.

$$f(x_1, x_2) = sin(x_2)exp(1 - cos(x_1))^2 + cos(x_1)exp(1 - sin(x_2))^2 + (x_1 - x_2)^2$$

Para la región acotada entre:

$$-10 \leq x_1 \leq 0$$

$$-6.5 \leq x_2 \leq 0$$

la función tiene múltiples mínimos locales y un único el mínimo global que se encuentra en:

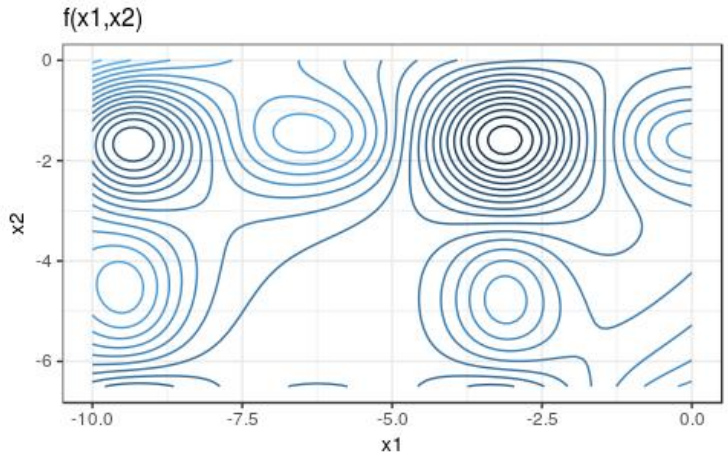
$$f(-3.1302468, -1.5821422) = -106.7645367$$

Función objetivo

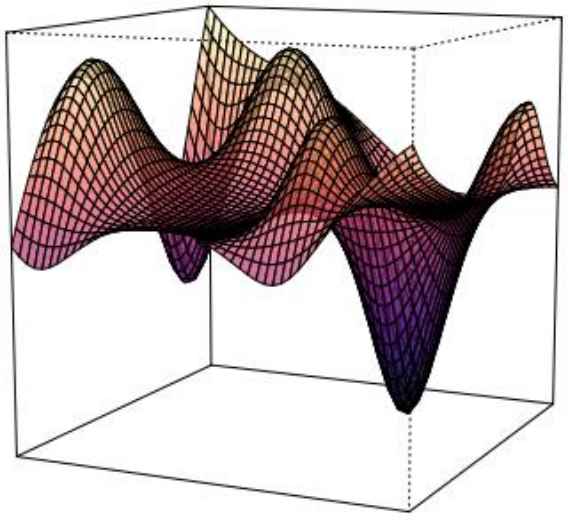
```
# Función objetivo a optimizar.
funcion <- function(x1, x2){
  sin(x2)*exp(1-cos(x1))^2 + cos(x1)*exp(1-sin(x2))^2 + (x1-x2)^2
}
```

Representación gráfica de la función.

```
x1 <- seq(-10, 0, length.out = 100)
x2 <- seq(-6.5, 0, length.out = 100)

datos <- expand.grid(x1 = x1, x2 = x2)
datos <- datos %>%
         mutate(f_x = map2_dbl(x1, x2, .f = funcion))

ggplot(data = datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  labs(title = "f(x1,x2)") +
  theme_bw() +
  theme(legend.position = "none",
        title = element_text(size = 10))
```




```
x1  <- seq(-10, 0, length.out = 50)
x2  <- seq(-6.5, 0, length.out = 50)
f_x <- outer(x1, x2, FUN = funcion)

library(viridis)
colores        <- viridis::magma(n = 100, alpha = 0.7)
z.facet.center <- (f_x[-1, -1] + f_x[-1, -ncol(f_x)] +
                     f_x[-nrow(f_x), -1] +
                     f_x[-nrow(f_x), -ncol(f_x)])/4
z.facet.range  <- cut(z.facet.center, 100)

par(mai = c(0,0,0,0))
persp(x = x1, y = x2, z = f_x, shade = 0.8, r = 8,
      phi = 10, theta = 25, col = colores[z.facet.range],
      axes = FALSE)
```



Optimización

```
# Ejemplo ilustrativo, en un caso real, emplear como mínimo 100 individuos por
# generación.
resultados_ga <- optimizar_ga(
                 funcion_objetivo = funcion,
                 n_variables = 2,
                 optimizacion = "minimizar",
                 limite_inf = c(-10, -5.6),
                 limite_sup = c(0, 0),
                 n_poblacion = 30,
                 n_generaciones = 500,
                 elitismo = 0.01,
                 prob_mut = 0.1,
                 distribucion = "uniforme",
                 min_distribucion = -1,
                 max_distribucion = 1,
                 metodo_seleccion = "tournament",
                 parada_temprana = TRUE,
                 rondas_parada = 10,
                 tolerancia_parada = 0.0001,
                 verbose = FALSE
                 )
```

```
## [1] "Algoritmo detenido en la generacion 21 por falta cambio mínimo de 1e-04 dur
ante 10 generaciones consecutivas."
```

Resultados

El objeto devuelto por la función `optimizar_ga` almacena la información (*fitness*, valor de las variables,...) del mejor individuo de cada generación.

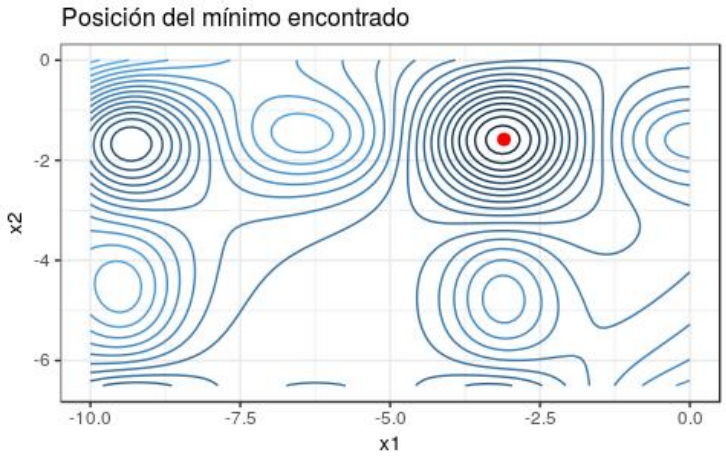
```
resultados_ga$df_resultados
```

Mejor individuo

```
resultados_ga$mejor_individuo
```

```
## [1] -3.101119 -1.582946
```

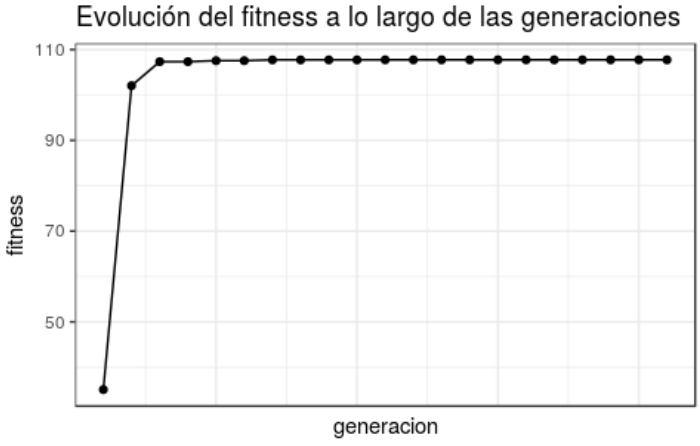
```
ggplot(data= datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  annotate(geom = "point",
           x = resultados_ga$mejor_individuo[1],
           y =  resultados_ga$mejor_individuo[2],
           color = "red", size = 2.5) +
  labs(title = "Posición del mínimo encontrado") +
  theme_bw() + theme(legend.position = "none", title = element_text(size = 10))
```



Evolución del error

En el siguiente gráfico se puede ver cómo evoluciona el *fitness* del mejor individuo a medida que avanzan las generaciones.

```
library(ggplot2)
ggplot(data = resultados_ga$df_resultados,
       aes(x = generacion, y = fitness)) +
  geom_line(aes(group = 1)) +
  geom_point() +
  labs(title = "Evolución del fitness a lo largo de las generaciones") +
  theme_bw() + theme(axis.text.x = element_blank(), axis.ticks.x = element_blank())
```



Animación de cómo avanza la búsqueda del mínimo (para formato html)

```
library(gganimate)
evolucion_resultados <- resultados_ga$df_resultados %>%
                        separate(col = predictores, sep = ",",
                                 into = c("x1", "x2")) %>%
                        mutate(x1 = as.numeric(x1),
                               x2 = as.numeric(x2)) %>%
                        select(generacion, x1, x2)

gift <- ggplot() +
        geom_contour(data = datos,
                     aes(x = x1, y = x2, z = f_x, colour = stat(level)),
                     bins = 20) +
        geom_point(data = evolucion_resultados,
                   aes(x = x1, y = x2),
                   color = "red",
                   size = 2) +
        theme_bw() +
        theme(legend.position = "none",
              title = element_text(size = 10)) +
        transition_manual(frames = generacion) +
        ggtitle("Posición del mínimo encontrado",
                subtitle = "Generación {frame}")

animate(plot = gift)
```

Ejemplo 3

En este ejemplo se pretende evaluar la capacidad del algoritmo genético para encontrar el mínimo de la función de *Ackley*.

$$f(x_1, x_2) = -20exp[-0.2\sqrt{0.5(x_1^2 + x_2^2)}] - exp[0.5(\cos 2\pi x_1 + \cos 2\pi x_2)] + e + 20$$

la función tiene múltiples mínimos locales y un único el mínimo global que se encuentra en:

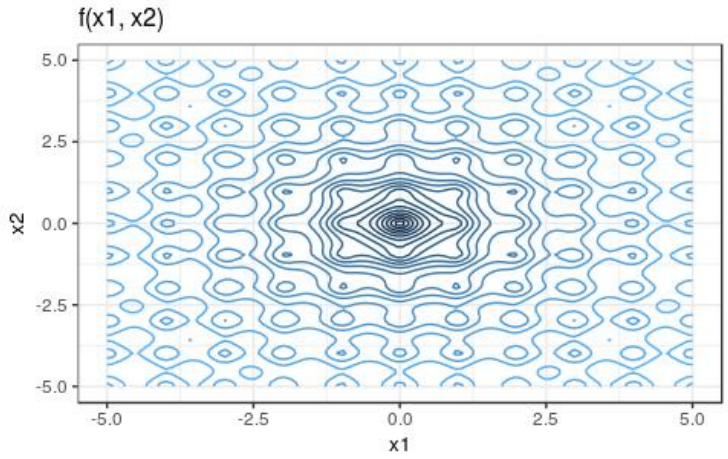
$$f(0,0) = 0$$

Función objetivo

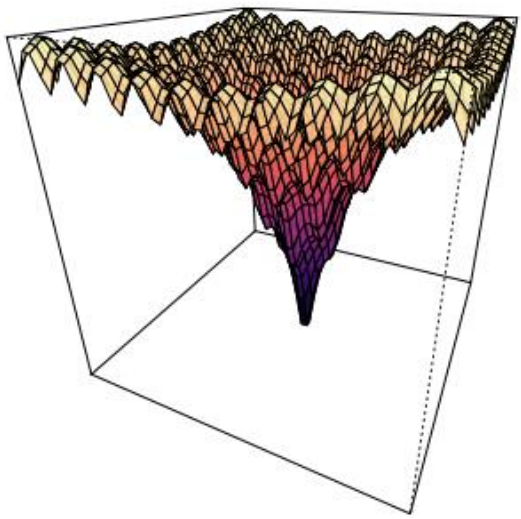
```
# Función objetivo a optimizar.
funcion <- function(x1, x2){
  -20*exp(-0.7*sqrt(0.5*(x1^2 + x2^2))) - exp(0.5*(cos(2*pi*x1) + cos(2*pi*x2))) +
exp(1) + 20
}
```

Representación gráfica de la función.

```
x1 <- seq(-5, 5, length.out = 100)
x2 <- seq(-5, 5, length.out = 100)

datos <- expand.grid(x1 = x1, x2 = x2)
datos <- datos %>%
         mutate(f_x = map2_dbl(x1, x2, .f = funcion))

ggplot(data = datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  labs(title = "f(x1, x2)") +
  theme_bw() +
  theme(legend.position = "none",
        title = element_text(size = 10))
```




```r
x1  <- seq(-5, 5, length.out = 50)
x2  <- seq(-5, 5, length.out = 50)
f_x <- outer(x1, x2, FUN = funcion)

library(viridis)
colores        <- viridis::magma(n = 100, alpha = 0.9)
z.facet.center <- (f_x[-1, -1] + f_x[-1, -ncol(f_x)] +
                     f_x[-nrow(f_x), -1] +
                     f_x[-nrow(f_x), -ncol(f_x)])/4
z.facet.range  <- cut(z.facet.center, 100)

par(mai = c(0,0,0,0))
persp(x = x1, y = x2, z = f_x, shade = 0.8,
      r = 1, phi = 25, theta = 25,
      col = colores[z.facet.range], axes = FALSE)
```



Optimización

```
resultados_ga <- optimizar_ga(
                 funcion_objetivo = funcion,
                 n_variables = 2,
                 optimizacion = "minimizar",
                 limite_inf = c(-5, -5),
                 limite_sup = c(5, 5),
                 n_poblacion = 200,
                 n_generaciones = 1000,
                 elitismo = 0.01,
                 prob_mut = 0.1,
                 distribucion = "uniforme",
                 min_distribucion = -1,
                 max_distribucion = 1,
                 metodo_seleccion = "tournament",
                 parada_temprana = TRUE,
                 rondas_parada = 10,
                 tolerancia_parada = 10^-8,
                 verbose = FALSE
                 )
```

```
## [1] "Algoritmo detenido en la generacion 36 por falta cambio mínimo de 1e-08 dur
ante 10 generaciones consecutivas."
```

Resultados

El objeto devuelto por la función `optimizar_ga` almacena la información (*fitness*, valor de las variables,...) del mejor individuo de cada generación.

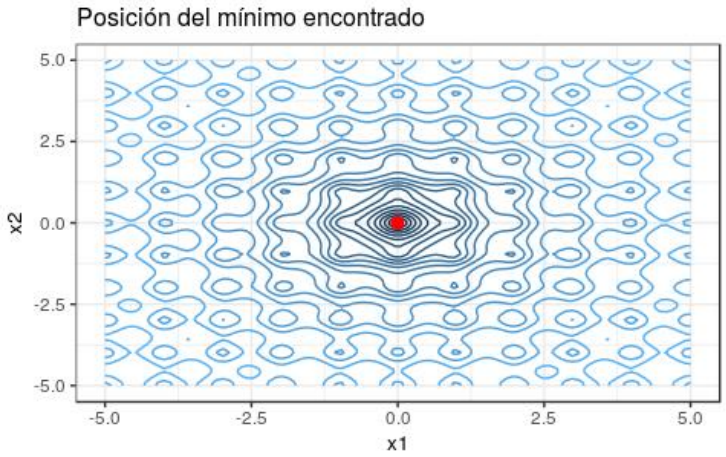
```
resultados_ga$df_resultados
```

Mejor individuo

```
resultados_ga$mejor_individuo
```

```
## [1] -0.0063901665  0.0005420274
```

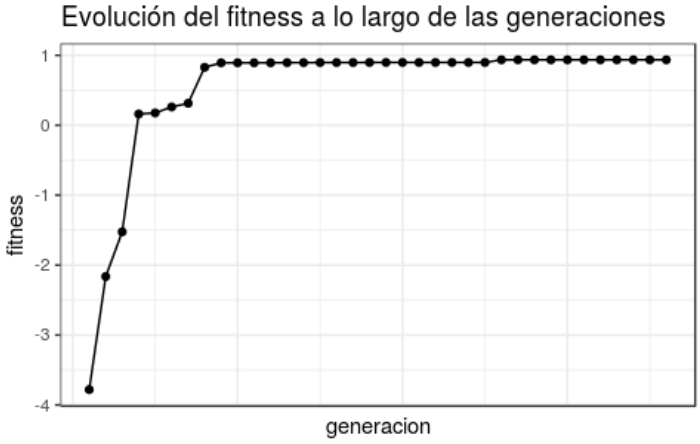
```
ggplot(data= datos, aes(x = x1, y = x2, z = f_x)) +
  geom_contour(aes(colour = stat(level)), bins = 20) +
  annotate(geom = "point",
           x = resultados_ga$mejor_individuo[1],
           y =  resultados_ga$mejor_individuo[2],
           color = "red", size = 2.5) +
  labs(title = "Posición del mínimo encontrado") +
  theme_bw() + theme(legend.position = "none", title = element_text(size = 10))
```



Evolución del error

En el siguiente gráfico se puede ver cómo evoluciona el *fitness* del mejor individuo a medida que avanzan las generaciones.

```
library(ggplot2)
ggplot(data = resultados_ga$df_resultados,
       aes(x = generacion, y = fitness)) +
  geom_line(aes(group = 1)) +
  geom_point() +
  labs(title = "Evolución del fitness a lo largo de las generaciones") +
  theme_bw() + theme(axis.text.x = element_blank(), axis.ticks.x = element_blank())
```



Paralelización

Uno de los inconvenientes de los algoritmos genéticos es su alto requerimiento computacional. Por ejemplo, si se establecen 1000 generaciones con 150 individuos por generación, la función objetivo se evalúa $1000x150 = 150000$ veces.

Los ejemplos anteriores se ejecutan en poco tiempo por la sencillez de las funciones objetivo pero, en la práctica, suelen ser mucho más complejas. Dos de las estrategias que se pueden emplear para agilizar el proceso son:

- Parada temprana: detener el algoritmo si tras n generaciones consecutivas no se ha conseguido un cambio mínimo. Esta estrategia está implementada en los ejemplos anteriores.
- Paralelización:
  - Evaluar de forma simultánea varios individuos de la población empleando múltiples cores del ordenador.
  - Paralelizar la función objetivo en el caso de que pueda hacerse.

Versión paralelizada

ESTA IMPLEMENTACIÓN NO FUNCIONA EN WINDOWS

```
calcular_fitness_poblacion_paral <- function(poblacion, funcion_objetivo,
                                             optimizacion, n_cores = NULL,
                                             verbose = TRUE, ...) {
  # Esta función devuelve el fitness de cada individuo de una población.
  #
  # ARGUMENTOS
  # ============================================================================
  # poblacion: matriz que representa la población de individuos.
  # funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
  #                   sido definida previamente.
  # optimizacion:     "maximizar" o "minimizar". Dependiendo de esto, la relación
  #                   del fitness es directamente o indirectamente proporcional
  #                   al valor de la función.
  # verbose:          mostrar información del proceso por pantalla.
  #
  # RETORNO
  # ============================================================================
  #   vector con el fitness de todos los individuos de la población. El orden de
  #   los valores se corresponde con el orden de las filas de la matriz población.
```

```
  # Paquetes necesarios para paralelizar.
  library(furrr)
  library(purrr)

  if (is.null(n_cores)) {
    future::plan(
      strategy = future::multiprocess,
      workers = future::availableCores(constraints = "multicore") - 1
    )
  } else {
    future::plan(
      strategy = future::multiprocess,
      workers = n_cores
    )
  }

  # Se almacena cada individuo (cada fila de la matriz población) como un
  # elemento de una lista.
  poblacion <- purrr::map(
    .x = 1:nrow(poblacion),
    .f = function(i) {
      poblacion[i, ]
    }
  )

  # Se aplica la función "calcular_fitness_individuo" a cada elemento de la
  # lista población.
  fitness_poblacion <- furrr::future_map_dbl(
    .x = poblacion,
    .f = calcular_fitness_individuo,
    funcion_objetivo = funcion_objetivo,
    optimizacion = optimizacion,
    verbose = verbose
  )

  if (verbose) {
    print(paste(
      "Fitness calculado para los",
      nrow(poblacion),
      "individuos de la población."
    ))
    cat("\n")
  }

  return(unlist(fitness_poblacion))
}
```

Ejemplo

Se calcula el *fitness* de todos los individuos de una población formada por 5 individuos.

```
# Función objetivo a optimizar.
funcion <- function(x, y){
  return(x^2 + y^2)
}

# Población simulada.
poblacion <- crear_poblacion(
                n_poblacion = 5,
                n_variables = 2,
                limite_inf =  c(-10, -10),
                limite_sup =  c(+10, +10),
                verbose = TRUE)
```

```
## [1] "Población inicial creada"
## [1] "Número de individuos = 5"
## [1] "Límites inferiores de cada variable: -10, -10"
## [1] "Límites superiores de cada variable: 10, 10"
```

```
# Cálculo del fitness de todos los individuos.
fitness_poblacion <- calcular_fitness_poblacion_paral(
                        poblacion = poblacion,
                        funcion_objetivo = funcion,
                        optimizacion = "minimizar",
                        verbose = TRUE
                      )
```

```
## [1] "El fitness calculado para minimizar es de: -51.9291043145496"
##
## [1] "El fitness calculado para minimizar es de: -74.5756694471396"
##
## [1] "El fitness calculado para minimizar es de: -88.6337452232735"
##
## [1] "El fitness calculado para minimizar es de: -162.419131387805"
##
## [1] "El fitness calculado para minimizar es de: -152.915101724577"
##
## [1] "Fitness calculado para los  individuos de la población."
```

```
fitness_poblacion
```

```
## [1]  -51.92910  -74.57567  -88.63375 -162.41913 -152.91510
```

Para incluir la opción de paralelizado, se repite la función del algoritmo completo, esta vez, incluyendo el argumento paralelizado con el que el usuario pueda especificar que se emplee la función `calcular_fitness_poblacion` o `calcular_fitness_poblacion_paral`.

```
optimizar_ga <- function(
                         funcion_objetivo,
                         n_variables,
                         optimizacion,
                         limite_inf = NULL,
                         limite_sup = NULL,
                         n_poblacion = 20,
                         n_generaciones = 10,
                         elitismo = 0.1,
                         prob_mut = 0.01,
                         distribucion = "uniforme",
                         media_distribucion = 1,
                         sd_distribucion = 1,
                         min_distribucion = -1,
                         max_distribucion = 1,
                         metodo_seleccion = "tournament",
                         parada_temprana = FALSE,
                         rondas_parada = NULL,
                         tolerancia_parada = NULL,
                         paralelizado = FALSE,
                         n_cores = NULL,
                         verbose = FALSE,
                         ...) {

  # ARGUMENTOS:
  # ============================================================================
  #
  # funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
  #                   sido definida previamente.
  # n_variables:      longitud de los individuos.
  # optimizacion:     "maximizar" o "minimizar". Dependiendo de esto, la relación
  #                   del fitness es directamente o indirectamente proporcional al
  #                   valor de la función.
  # limite_inf:       vector con el límite inferior de cada variable. Si solo se
  #                   quiere imponer límites a algunas variables, emplear NA para
  #                   las que no se quiere acotar.
  # limite_sup:       vector con el límite superior de cada variable. Si solo se
  #                   quiere imponer límites a algunas variables, emplear NA para
  #                   las que no se quieren acotar.
  # n_poblacion:      número total de individuos de la población.
  # n_generaciones:   número total de generaciones creadas.
  # elitismo:         porcentaje de mejores individuos de la población actual que
  #                   pasan directamente a la siguiente población.
  # prob_mut:         probabilidad que tiene cada posición del individuo de mutar.
  # distribucion:     distribución de la que obtener el factor de mutación. Puede
  #                   ser: "normal", "uniforme" o "aleatoria".
```

```
# media_distribucion:media de la distribución si se selecciona distribucion="normal".
# sd_distribucion:  desviación estándar de la distribución si se selecciona
#                   distribucion="normal".
# min_distribucion: mínimo la distribución si se selecciona distribucion="uniforme".
# max_distribucion: máximo la distribución si se selecciona distribucion="uniforme".
# metodo_seleccion: método para establecer la probabilidad de selección. Puede
#                   ser: "ruleta", "rank" o "tournament".
# parada_temprana:  si durante las últimas "rondas_parada" generaciones la diferencia
#                   absoluta entre mejores individuos no es superior al valor de
#                  "tolerancia_parada", se detiene el algoritmo y no se crean
#                   nuevas generaciones.
# rondas_parada:    número de generaciones consecutivas sin mejora mínima para que
#                   se active la parada temprana.
# tolerancia_parada: valor mínimo que debe tener la diferencia de generaciones
#                    consecutivas para considerar que hay cambio.
# paralelizado:     TRUE para paralelizar el algoritmo genético.
# n_cores:          número de cores para la paralelización.
# verbose:          TRUE para que se imprima por pantalla el resultado de cada
#                   paso del algoritmo.

# RETORNO:
# ============================================================================
# La función devuelve una lista con 5 elementos:
# fitness:            una lista con el fitness del mejor individuo de cada
#                     generación.
# mejores_individuos: una lista con la combinación de predictores del mejor
#                     individuo de cada generación.
# mejor_individuo:    combinación de predictores del mejor individuo encontrado
#                     en todo el proceso.
# diferencia_abs:     una lista con la diferencia absoluta entre el fitness
#                     del mejor individuo de generaciones consecutivas.
# df_resultados:      un dataframe con todos los resultados anteriores.



# COMPROBACIONES INICIALES
# ============================================================================

# Si se activa la parada temprana, hay que especificar los argumentos
# rondas_parada y tolerancia_parada.
if(isTRUE(parada_temprana)&&(is.null(rondas_parada)|is.null(tolerancia_parada))){
  stop(paste(
    "Para activar la parada temprana es necesario indicar un valor",
    "de rondas_parada y de tolerancia_parada."
  ))
}

# ESTABLECER LOS LÍMITES DE BÚSQUEDA SI EL USUARIO NO LO HA HECHO
# ============================================================================
if (is.null(limite_sup) | is.null(limite_inf)) {
  warning(paste(
```

```
    "Es altamente recomendable indicar los límites dentro de los",
    "cuales debe buscarse la solución de cada variable.",
    "Por defecto se emplea: [-10^3, 10^3]."
  ))
}

if (any(
      is.null(limite_sup), is.null(limite_inf), any(is.na(limite_sup)),
      any(is.na(limite_inf))
)) {
  warning(paste(
    "Los límites empleados por defecto cuando no se han definido son:",
    " [-10^3, 10^3]."
  ))
  cat("\n")
}
# Si no se especifica limite_inf, el valor mínimo que pueden tomar las variables
# es -10^3.
if (is.null(limite_inf)) {
  limite_inf <- rep(x = -10^3, times = n_variables)
}
# Si no se especifica limite_sup, el valor máximo que pueden tomar las variables
# es 10^3.
if (is.null(limite_sup)) {
  limite_sup <- rep(x = 10^3, times = n_variables)
}
# Si los límites no son nulos, se reemplazan aquellas posiciones NA por el valor
# por defecto -10^3 y 10^3
if (!is.null(limite_inf)) {
  limite_inf[is.na(limite_inf)] <- -10^3
}

if (!is.null(limite_sup)) {
  limite_sup[is.na(limite_sup)] <- 10^3
}

# ALMACENAMIENTO DE RESULTADOS
# ==============================================================================
# Por cada generación se almacena el mejor individuo, su fitness, y el porcentaje
# de mejora respecto a la última generación.
resultados_fitness <- vector(mode = "list", length = n_generaciones)
resultados_individuo <- vector(mode = "list", length = n_generaciones)
diferencia_abs <- vector(mode = "list", length = n_generaciones)



# CREACIÓN DE LA POBLACIÓN INICIAL
# ==============================================================================
poblacion <- crear_poblacion(
  n_poblacion = n_poblacion,
```

```
    n_variables = n_variables,
    limite_inf = limite_inf,
    limite_sup = limite_sup,
    verbose = verbose
  )
  # ITERACIÓN DE POBLACIONES
  # ============================================================================
  for (i in 1:n_generaciones) {
    if (verbose) {
      print("----------------------")
      print(paste("Generación:", i))
      print("----------------------")
    }

    # CALCULAR FITNESS DE LOS INDIVIDUOS DE LA POBLACIÓN
    # ============================================================================
    if (!paralelizado) {
      fitness_ind_poblacion <- calcular_fitness_poblacion(
        poblacion = poblacion,
        funcion_objetivo = funcion_objetivo,
        optimizacion = optimizacion,
        verbose = verbose
      )
    }

    if (paralelizado) {
      fitness_ind_poblacion <- calcular_fitness_poblacion_paral(
        poblacion = poblacion,
        funcion_objetivo = funcion_objetivo,
        optimizacion = optimizacion,
        verbose = verbose,
        n_cores = n_cores
      )
    }

    # SE ALMACENA EL MEJOR INDIVIDUO DE LA POBLACIÓN ACTUAL
    # ============================================================================
    fitness_mejor_individuo <- max(fitness_ind_poblacion)
    mejor_individuo <- poblacion[which.max(fitness_ind_poblacion), ]
    resultados_fitness[[i]] <- fitness_mejor_individuo
    resultados_individuo[[i]] <- mejor_individuo

    # SE CALCULA LA DIFERENCIA ABSOLUTA RESPECTO A LA GENERACIÓN ANTERIOR
    # ============================================================================
    # La diferencia solo puede calcularse a partir de la segunda generación.
    if (i > 1) {
      diferencia_abs[[i]]<-abs(resultados_fitness[[i-1]]-resultados_fitness[[i]])
    }
```

```r
# NUEVA POBLACIÓN
# ============================================================================
nueva_poblacion <- matrix(
  data = NA,
  nrow = nrow(poblacion),
  ncol = ncol(poblacion)
)

# ELITISMO
# ============================================================================
# El elitismo indica el porcentaje de mejores individuos de la población
# actual que pasan directamente a la siguiente población. De esta forma, se
# asegura que, la siguiente generación, no sea nunca inferior.

if (elitismo > 0) {
  n_elitismo <- ceiling(nrow(poblacion) * elitismo)
  posicion_n_mejores <- order(fitness_ind_poblacion, decreasing = TRUE)
  posicion_n_mejores <- posicion_n_mejores[1:n_elitismo]
  nueva_poblacion[1:n_elitismo, ] <- poblacion[posicion_n_mejores, ]
} else {
  n_elitismo <- 0
}

# CREACIÓN DE NUEVOS INDIVIDUOS POR CRUCES
# ============================================================================
for (j in (n_elitismo + 1):nrow(nueva_poblacion)) {
  # Seleccionar parentales
  indice_parental_1 <- seleccionar_individuo(
    vector_fitness = fitness_ind_poblacion,
    metodo_seleccion = metodo_seleccion
  )
  indice_parental_2 <- seleccionar_individuo(
    vector_fitness = fitness_ind_poblacion,
    metodo_seleccion = metodo_seleccion
  )
  parental_1 <- poblacion[indice_parental_1, ]
  parental_2 <- poblacion[indice_parental_2, ]
  # Cruzar parentales para obtener la descendencia
  descendencia <- cruzar_individuos(
    parental_1 = parental_1,
    parental_2 = parental_2
  )
  # Mutar la descendencia
  descendencia <- mutar_individuo(
    individuo = descendencia,
    prob_mut = prob_mut,
    limite_inf = limite_inf,
    limite_sup = limite_sup,
    distribucion = distribucion,
    media_distribucion = media_distribucion,
```

```
      sd_distribucion = sd_distribucion,
      min_distribucion = min_distribucion,
      max_distribucion = max_distribucion
    )

    nueva_poblacion[j, ] <- descendencia
  }
  poblacion <- nueva_poblacion

  # CRITERIO DE PARADA
  # ============================================================================
  # Si durante las últimas n generaciones la diferencia absoluta entre mejores
  # individuos no es superior al valor de tolerancia_parada, se detiene el
  #  algoritmo y no se crean nuevas generaciones.
  if (parada_temprana && (i > rondas_parada)) {
    ultimos_n <- tail(unlist(diferencia_abs), n = rondas_parada)
    if (all(ultimos_n < tolerancia_parada)) {
      print(paste(
        "Algoritmo detenido en la generacion", i,
        "por falta cambio mínimo de", tolerancia_parada,
        "durante", rondas_parada,
        "generaciones consecutivas."
      ))
      break()
    }
  }
}

# IDENTIFICACIÓN DEL MEJOR INDIVIDUO DE TODO EL PROCESO
# ============================================================================
mejor_individuo <- resultados_individuo[[which.max(unlist(resultados_fitness))]]

# RESULTADOS
# ============================================================================
# Para crear el dataframe se convierten las listas a vectores del mismo tamaño.
fitness <- unlist(resultados_fitness)
predictores <- resultados_individuo[!sapply(resultados_individuo, is.null)]
predictores <- sapply(predictores, function(x) {
                                      paste(x, collapse = ", ")
                                    }
                      )
diferencia_abs <- c(NA, unlist(diferencia_abs))

df_resultados <- data.frame(
  generacion = seq_along(fitness),
  fitness = fitness,
  predictores = predictores,
  diferencia_abs = diferencia_abs
)
```

```
  return(list(
    fitness = resultados_fitness,
    mejores_individuos = resultados_individuo,
    diferencia_abs = diferencia_abs,
    df_resultados = df_resultados,
    mejor_individuo = mejor_individuo
  ))
}
```

Comparación

Se compara el tiempo necesario para ejecutar la optimización empleando paralelización y sin ella. Como la paralelización afecta al paso en el que se calcula el *fitness* de todos los individuos de la población, la diferencia de tiempo se notará más cuanto mayor sea el tamaño de la población, pero no se verá afectado por el número de generaciones. Para este ejemplo se emplea `n_poblacion = 1000`, `n_generaciones = 100` y se desactiva la parada temprana para que en ambos casos se ejecuten el mismo número de iteraciones.

Función objetivo

```
# Función objetivo a optimizar.
funcion <- function(x, y){
  sin(y)*exp(1-cos(x))^2 + cos(x)*exp(1-sin(y))^2 + (x-y)^2
}
```

Sin paralelización

```
library(tictoc)

tic()
resultados <- optimizar_ga(
                  funcion_objetivo = funcion,
                  n_variables = 2,
                  optimizacion = "minimizar",
                  limite_inf = c(-10, -5),
                  limite_sup = c(0, 0),
                  n_poblacion = 1000,
                  n_generaciones = 100,
                  elitismo = 0,
                  prob_mut = 0.1,
                  distribucion = "aleatoria",
                  media_distribucion = 1,
```

```
                  sd_distribucion = 1,
                  min_distribucion = -1,
                  max_distribucion = 1,
                  verbose = FALSE,
                  parada_temprana = FALSE,
                  rondas_parada = NULL,
                  tolerancia_parada = NULL,
                  paralelizado = FALSE
                  )
toc()
```

```
## 9.654 sec elapsed
```

Con paralelización

```
library(tictoc)

tic()
resultados <- optimizar_ga(
                  funcion_objetivo = funcion,
                  n_variables = 2,
                  optimizacion = "minimizar",
                  limite_inf = c(-10, -5),
                  limite_sup = c(0, 0),
                  n_poblacion = 1000,
                  n_generaciones = 100,
                  elitismo = 0,
                  prob_mut = 0.1,
                  distribucion = "aleatoria",
                  media_distribucion = 1,
                  sd_distribucion = 1,
                  min_distribucion = -1,
                  max_distribucion = 1,
                  verbose = FALSE,
                  parada_temprana = FALSE,
                  rondas_parada = NULL,
                  tolerancia_parada = NULL,
                  paralelizado = TRUE
                  )
toc()
```

```
## 16.804 sec elapsed
```

Puede observarse que, para una función tan sencilla de calcular, la latencia incorporada por el proceso de paralelización no compensa.

Algoritmo genético y Nelder-Mead

Los métodos de optimización basados en algoritmo genético son buenos identificando áreas de mínimos globales ya que exploran una gran parte del dominio de la función, sin embargo, una vez que se encuentran en una región local, son poco eficientes aproximándose al mínimo en comparación a otros métodos de optimización local como el de Nelder-Mead Simplex.

En el siguiente ejemplo, se pretende minimizar la función $f(x_1, x_2) = x_1^2 + x_2^2$. Esta función tiene un único mínimo local ($x_1 = 0, x_2 = 0$) que conincide con el mínimo global, por lo que pueden emplearse tanto métodos de optimización global (algoritmo genético) como métodos de optimización local (Nelder-Mead Simplex). Véase la rapidez y precisión con la que encuentran cada uno la solución.

Función objetivo

```
# Función objetivo a optimizar. Mínimo global en (0,0).
funcion <- function(x1, x2){
  return(x1^2 + x2^2)
}
```

Optimización algoritmo genético

```
tic()
resultados_ga <- optimizar_ga(
                   funcion_objetivo = funcion,
                   n_variables = 2,
                   optimizacion = "minimizar",
                   limite_inf = c(-10, -10),
                   limite_sup = c(10, 10),
                   n_poblacion = 150,
                   n_generaciones = 500,
                   elitismo = 0.01,
                   prob_mut = 0.1,
                   distribucion = "uniforme",
                   min_distribucion = -1,
                   max_distribucion = 1,
                   metodo_seleccion = "tournament",
                   parada_temprana = TRUE,
                   rondas_parada = 10,
                   tolerancia_parada = 10^-8,
                   verbose = FALSE
                   )
```

```
## [1] "Algoritmo detenido en la generacion 39 por falta cambio mínimo de 1e-08 dur
ante 10 generaciones consecutivas."
```

```
toc()
```

```
## 0.472 sec elapsed
```

```
resultados_ga$mejor_individuo
```

```
## [1]  0.0002143648 -0.0015953435
```

Optimización Nelder-Mead

```
tic()
optim(par = c(10, 10),
      fn = function(par){
              do.call(funcion, args = as.list(par))
           },
      method = "Nelder-Mead")
```

```
## $par
## [1] 0.0003754010 0.0005179101
##
## $value
## [1] 4.091568e-07
##
## $counts
## function gradient
##       63       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
toc()
```

```
## 0.005 sec elapsed
```

El método de optimización local Nelder-Mead converge, en mucho menos tiempo, en un valor más próximo al mínimo real.

Combinación de métodos

Dado que ninguno de los dos métodos, algoritmo genético y Nelder-Mead, necesitan las derivadas de la función objetivo, pueden combinarse para conseguir la capacidad de encontrar áreas de mínimos globales del primero y la rápida convergencia local del segundo.

La siguiente función aplica primero un algoritmo genético y después intenta mejorar el resultado final con el algoritmo Nelder-Mead.

```
optimizar_ga <- function(
                         funcion_objetivo,
                         n_variables,
                         optimizacion,
                         limite_inf = NULL,
                         limite_sup = NULL,
                         n_poblacion = 20,
                         n_generaciones = 10,
                         elitismo = 0.1,
                         prob_mut = 0.01,
                         distribucion = "uniforme",
                         media_distribucion = 1,
                         sd_distribucion = 1,
                         min_distribucion = -1,
                         max_distribucion = 1,
                         metodo_seleccion = "tournament",
                         parada_temprana = FALSE,
                         rondas_parada = NULL,
                         tolerancia_parada = NULL,
                         Nelder_Mead = TRUE,
                         paralelizado = FALSE,
                         n_cores = NULL,
                         verbose = FALSE,
                         ...) {
  # ARGUMENTOS:
  # ============================================================================
  #
  # funcion_objetivo: nombre de la función que se desea optimizar. Debe de haber
  #                   sido definida previamente.
  # n_variables:      longitud de los individuos.
  # optimizacion:     "maximizar" o "minimizar". Dependiendo de esto, la relación
  #                   del fitness es directamente o indirectamente proporcional al
  #                   valor de la función.
  # limite_inf:       vector con el límite inferior de cada variable. Si solo se
  #                   quiere imponer límites a algunas variables, emplear NA para
  #                   las que no se quiere acotar.
  # limite_sup:       vector con el límite superior de cada variable. Si solo se
  #                   quiere imponer límites a algunas variables, emplear NA para
  #                   las que no se quieren acotar.
```

```
  # n_poblacion:      número total de individuos de la población.
  # n_generaciones:   número total de generaciones creadas.
  # elitismo:         porcentaje de mejores individuos de la población actual que
  #                   pasan directamente a la siguiente población.
  # prob_mut:         probabilidad que tiene cada posición del individuo de mutar.
  # distribucion:     distribución de la que obtener el factor de mutación. Puede
  #                   ser: "normal", "uniforme" o "aleatoria".
  # media_distribucion:media de la distribución si se selecciona distribucion="normal".
  # sd_distribucion:  desviación estándar de la distribución si se selecciona
  #                   distribucion="normal".
  # min_distribucion: mínimo la distribución si se selecciona distribucion="uniforme".
  # max_distribucion: máximo la distribución si se selecciona distribucion="uniforme".
  # metodo_seleccion: método para establecer la probabilidad de selección. Puede
  #                   ser: "ruleta", "rank" o "tournament".
  # parada_temprana:  si durante las últimas "rondas_parada" generaciones la diferencia
  #                   absoluta entre mejores individuos no es superior al valor de
  #                  "tolerancia_parada", se detiene el algoritmo y no se crean
  #                   nuevas generaciones.
  # rondas_parada:    número de generaciones consecutivas sin mejora mínima para que
  #                   se active la parada temprana.
  # tolerancia_parada: valor mínimo que debe tener la diferencia de generaciones
  #                    consecutivas para considerar que hay cambio.
  # Nelder_Mead:      TRUE para que el mejor individuo devuelto por el algoritmo
  #                   genético se intente mejorar con optimización "Nelder_Mead".
  # paralelizado:     TRUE para paralelizar el algoritmo genético.
  # n_cores:          número de cores para la paralelización.
  # verbose:          TRUE para que se imprima por pantalla el resultado de cada
  #                   paso del algoritmo.

  # RETORNO:
  # ============================================================================
  # La función devuelve una lista con 5 elementos:
  # fitness:            una lista con el fitness del mejor individuo de cada
  #                     generación.
  # mejores_individuos: una lista con la combinación de predictores del mejor
  #                     individuo de cada generación.
  # mejor_individuo:    combinación de predictores del mejor individuo encontrado
  #                     en todo el proceso.
  # diferencia_abs:     una lista con la diferencia absoluta entre el fitness
  #                     del mejor individuo de generaciones consecutivas.
  # df_resultados:      un dataframe con todos los resultados anteriores.

  # COMPROBACIONES INICIALES
  # ============================================================================
  # Si se activa la parada temprana, hay que especificar los argumentos
  # rondas_parada y tolerancia_parada.
  if(isTRUE(parada_temprana)&&(is.null(rondas_parada)|is.null(tolerancia_parada))){
    stop(paste(
      "Para activar la parada temprana es necesario indicar un valor",
      "de rondas_parada y de tolerancia_parada."
    ))
```

```
}

# ESTABLECER LOS LÍMITES DE BÚSQUEDA SI EL USUARIO NO LO HA HECHO
# ==============================================================================
if (is.null(limite_sup) | is.null(limite_inf)) {
  warning(paste(
    "Es altamente recomendable indicar los límites dentro de los",
    "cuales debe buscarse la solución de cada variable.",
    "Por defecto se emplea: [-10^3, 10^3]."
  ))
}

if (any(
      is.null(limite_sup), is.null(limite_inf), any(is.na(limite_sup)),
      any(is.na(limite_inf))
)) {
  warning(paste(
    "Los límites empleados por defecto cuando no se han definido son:",
    " [-10^3, 10^3]."
  ))
  cat("\n")
}
# Si no se especifica limite_inf, el valor mínimo que pueden tomar las variables
# es -10^3.
if (is.null(limite_inf)) {
  limite_inf <- rep(x = -10^3, times = n_variables)
}
# Si no se especifica limite_sup, el valor máximo que pueden tomar las variables
# es 10^3.
if (is.null(limite_sup)) {
  limite_sup <- rep(x = 10^3, times = n_variables)
}
# Si los límites no son nulos, se reemplazan aquellas posiciones NA por el valor
# por defecto -10^3 y 10^3
if (!is.null(limite_inf)) {
  limite_inf[is.na(limite_inf)] <- -10^3
}

if (!is.null(limite_sup)) {
  limite_sup[is.na(limite_sup)] <- 10^3
}

# ALMACENAMIENTO DE RESULTADOS
# ==============================================================================
# Por cada generación se almacena el mejor individuo, su fitness, y el porcentaje
# de mejora respecto a la última generación.
resultados_fitness <- vector(mode = "list", length = n_generaciones)
resultados_individuo <- vector(mode = "list", length = n_generaciones)
diferencia_abs <- vector(mode = "list", length = n_generaciones)
```

```
# CREACIÓN DE LA POBLACIÓN INICIAL
# ============================================================================
poblacion <- crear_poblacion(
  n_poblacion = n_poblacion,
  n_variables = n_variables,
  limite_inf = limite_inf,
  limite_sup = limite_sup,
  verbose = verbose
)
# ITERACIÓN DE POBLACIONES
# ============================================================================
for (i in 1:n_generaciones) {
  if (verbose) {
    print("----------------------")
    print(paste("Generación:", i))
    print("----------------------")
  }

  # CALCULAR FITNESS DE LOS INDIVIDUOS DE LA POBLACIÓN
  # ============================================================================
  if (!paralelizado) {
    fitness_ind_poblacion <- calcular_fitness_poblacion(
      poblacion = poblacion,
      funcion_objetivo = funcion_objetivo,
      optimizacion = optimizacion,
      verbose = verbose
    )
  }

  if (paralelizado) {
    fitness_ind_poblacion <- calcular_fitness_poblacion_paral(
      poblacion = poblacion,
      funcion_objetivo = funcion_objetivo,
      optimizacion = optimizacion,
      verbose = verbose,
      n_cores = n_cores
    )
  }



  # SE ALMACENA EL MEJOR INDIVIDUO DE LA POBLACIÓN ACTUAL
  # ============================================================================
  fitness_mejor_individuo <- max(fitness_ind_poblacion)
  mejor_individuo <- poblacion[which.max(fitness_ind_poblacion), ]
  resultados_fitness[[i]] <- fitness_mejor_individuo
  resultados_individuo[[i]] <- mejor_individuo
```

```r
  # SE CALCULA LA DIFERENCIA ABSOLUTA RESPECTO A LA GENERACIÓN ANTERIOR
  # ============================================================================
  # La diferencia solo puede calcularse a partir de la segunda generación.
  if (i > 1) {
    diferencia_abs[[i]] <- abs(resultados_fitness[[i-1]]-resultados_fitness[[i]])
  }

  # NUEVA POBLACIÓN
  # ============================================================================
  nueva_poblacion <- matrix(
    data = NA,
    nrow = nrow(poblacion),
    ncol = ncol(poblacion)
  )

  # ELITISMO
  # ============================================================================
  # El elitismo indica el porcentaje de mejores individuos de la población
  # actual que pasan directamente a la siguiente población. De esta forma, se
  # asegura que, la siguiente generación, no sea nunca inferior.

  if (elitismo > 0) {
    n_elitismo <- ceiling(nrow(poblacion) * elitismo)
    posicion_n_mejores <- order(fitness_ind_poblacion, decreasing = TRUE)
    posicion_n_mejores <- posicion_n_mejores[1:n_elitismo]
    nueva_poblacion[1:n_elitismo, ] <- poblacion[posicion_n_mejores, ]
  } else {
    n_elitismo <- 0
  }

  # CREACIÓN DE NUEVOS INDIVIDUOS POR CRUCES
  # ============================================================================
  for (j in (n_elitismo + 1):nrow(nueva_poblacion)) {
    # Seleccionar parentales
    indice_parental_1 <- seleccionar_individuo(
      vector_fitness = fitness_ind_poblacion,
      metodo_seleccion = metodo_seleccion
    )
    indice_parental_2 <- seleccionar_individuo(
      vector_fitness = fitness_ind_poblacion,
      metodo_seleccion = metodo_seleccion
    )
    parental_1 <- poblacion[indice_parental_1, ]
    parental_2 <- poblacion[indice_parental_2, ]

    # Cruzar parentales para obtener la descendencia
    descendencia <- cruzar_individuos(
      parental_1 = parental_1,
      parental_2 = parental_2
```

```
      )
      # Mutar la descendencia
      descendencia <- mutar_individuo(
        individuo = descendencia,
        prob_mut = prob_mut,
        limite_inf = limite_inf,
        limite_sup = limite_sup,
        distribucion = distribucion,
        media_distribucion = media_distribucion,
        sd_distribucion = sd_distribucion,
        min_distribucion = min_distribucion,
        max_distribucion = max_distribucion
      )

      nueva_poblacion[j, ] <- descendencia
    }
    poblacion <- nueva_poblacion

    # CRITERIO DE PARADA
    # ============================================================================
    # Si durante las últimas n generaciones la diferencia absoluta entre mejores
    # individuos no es superior al valor de tolerancia_parada, se detiene el
    #  algoritmo y no se crean nuevas generaciones.

    if (parada_temprana && (i > rondas_parada)) {
      ultimos_n <- tail(unlist(diferencia_abs), n = rondas_parada)
      if (all(ultimos_n < tolerancia_parada)) {
        print(paste(
          "Algoritmo detenido en la generacion", i,
          "por falta cambio mínimo de", tolerancia_parada,
          "durante", rondas_parada,
          "generaciones consecutivas."
        ))
        break()
      }
    }
  }

  # IDENTIFICACIÓN DEL MEJOR INDIVIDUO DE TODO EL PROCESO
  # ============================================================================
  mejor_individuo <- resultados_individuo[[which.max(unlist(resultados_fitness))]]
  # Si Nelder_Mead = TRUE, el mejor individuo identificado mediante el
  # algoritmo genético se somete a un proceso de optimización Nelder-Mead con
  # el objetivo de mejorar la convergencia final.
  if (Nelder_Mead) {
    optmizacion_NM <- optim(
      par = mejor_individuo,
      fn = function(par) {
        do.call(funcion, args = as.list(par))
```

```
      },
      method = "Nelder-Mead"
    )
    mejor_individuo <- optmizacion_NM$par
  }

  # RESULTADOS
  # ============================================================================

  # Para crear el dataframe se convierten las listas a vectores del mismo tamaño.
  fitness <- unlist(resultados_fitness)
  predictores <- resultados_individuo[!sapply(resultados_individuo, is.null)]
  predictores <- sapply(predictores, function(x) {
    paste(x, collapse = ", ")
  })
  diferencia_abs <- c(NA, unlist(diferencia_abs))


  df_resultados <- data.frame(
    generacion = seq_along(fitness),
    fitness = fitness,
    predictores = predictores,
    diferencia_abs = diferencia_abs
  )

  return(list(
    fitness = resultados_fitness,
    mejores_individuos = resultados_individuo,
    diferencia_abs = diferencia_abs,
    df_resultados = df_resultados,
    mejor_individuo = mejor_individuo
  ))
}
```

Comparación

Función objetivo

```
# Función objetivo a optimizar. Mínimo global en (0,0).
funcion <- function(x1, x2){
  -20*exp(-0.7*sqrt(0.5*(x1^2 + x2^2))) - exp(0.5*(cos(2*pi*x1) + cos(2*pi*x2))) +
exp(1) + 20
}
```

Sin Nelder Mead

```
resultados_ga <- optimizar_ga(
                    funcion_objetivo = funcion,
                    n_variables = 2,
                    optimizacion = "minimizar",
                    limite_inf = c(-10, -10),
                    limite_sup = c(10, 10),
                    n_poblacion = 150,
                    n_generaciones = 500,
                    elitismo = 0.01,
                    prob_mut = 0.1,
                    distribucion = "uniforme",
                    min_distribucion = -1,
                    max_distribucion = 1,
                    metodo_seleccion = "tournament",
                    parada_temprana = TRUE,
                    rondas_parada = 10,
                    tolerancia_parada = 10^-8,
                    Nelder_Mead = FALSE,
                    paralelizado = FALSE,
                    verbose = FALSE
                    )
```

```
## [1] "Algoritmo detenido en la generacion 18 por falta cambio mínimo de 1e-08 dur
ante 10 generaciones consecutivas."
```

```
resultados_ga$mejor_individuo
```

```
## [1] 0.005356045 0.002459083
```

Con Nelder Mead

```
resultados_ga <- optimizar_ga(
                    funcion_objetivo = funcion,
                    n_variables = 2,
                    optimizacion = "minimizar",
                    limite_inf = c(-10, -10),
                    limite_sup = c(10, 10),
                    n_poblacion = 150,
                    n_generaciones = 500,
                    elitismo = 0.01,
                    prob_mut = 0.1,
                    distribucion = "uniforme",
                    min_distribucion = -1,
                    max_distribucion = 1,
                    metodo_seleccion = "tournament",
                    parada_temprana = TRUE,
                    rondas_parada = 10,
                    tolerancia_parada = 10^-8,
                    Nelder_Mead = TRUE,
                    paralelizado = FALSE,
                    verbose = FALSE
                    )
```

```
## [1] "Algoritmo detenido en la generacion 21 por falta cambio mínimo de 1e-08 dur
ante 10 generaciones consecutivas."
```

```
resultados_ga$mejor_individuo
```

```
## [1] 2.519614e-10 2.139588e-10
```

Empleando la combinación de ambos métodos de optimización se obtiene un resultado mucho más próximo al mínimo global comparado a si solo se emplea el algoritmo genético.

Bibliografía

John McCall, Genetic algorithms for modelling and optimisation, Journal of Computational and Applied Mathematics, Volume 184, Issue 1, 2005

Optimizing with GeneticAlgorithmsby Benjamin J. Lynch Feb 23, 2006

Durand, Nicolas & Alliot, Jean-Marc. (1999). A Combined Nelder-Mead Simplex and Genetic Algorithm.

Abdel-Rahman Hedar & Masao Fukushima (2003) Minimizing multimodal functions by simplex coding genetic algorithm, Optimization Methods and Software.

https://en.wikipedia.org/wiki/Genetic_algorithm

https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

https://www.sfu.ca/~ssurjano/optimization.html