
# Posgrado en Ciencias de la Tierra - IGF - UNAM

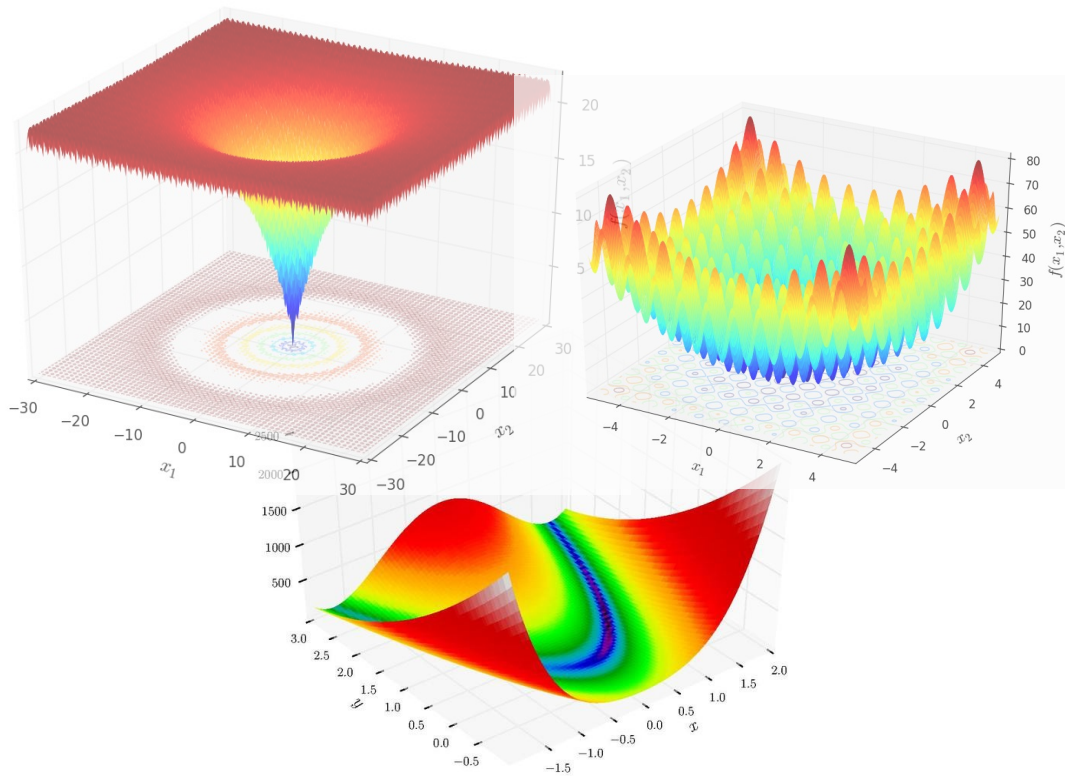
## Ayuda del programa *Copula cosimulation*

Autores:
Yarilis Gómez Martínez
Martín A. Díaz Viera

3 de diciembre de 2020

# Contenido





Este programa fue realizado con el propósito de implementar una metodología de cosimulación espacial basado en cópulas. Se necesita primeramente conocer un modelo de dependencia (función conjunta) y un variograma teórico que puede ser estimados a partir de datos de referencia (pozo o área cercana). Estos serán los datos de entrada para optimizar una función multiobjetivo con dos métodos de optimización (Evolución Diferencial y Recocido simulado generalizado) y poder simular una variable a partir de otra de apoyo.

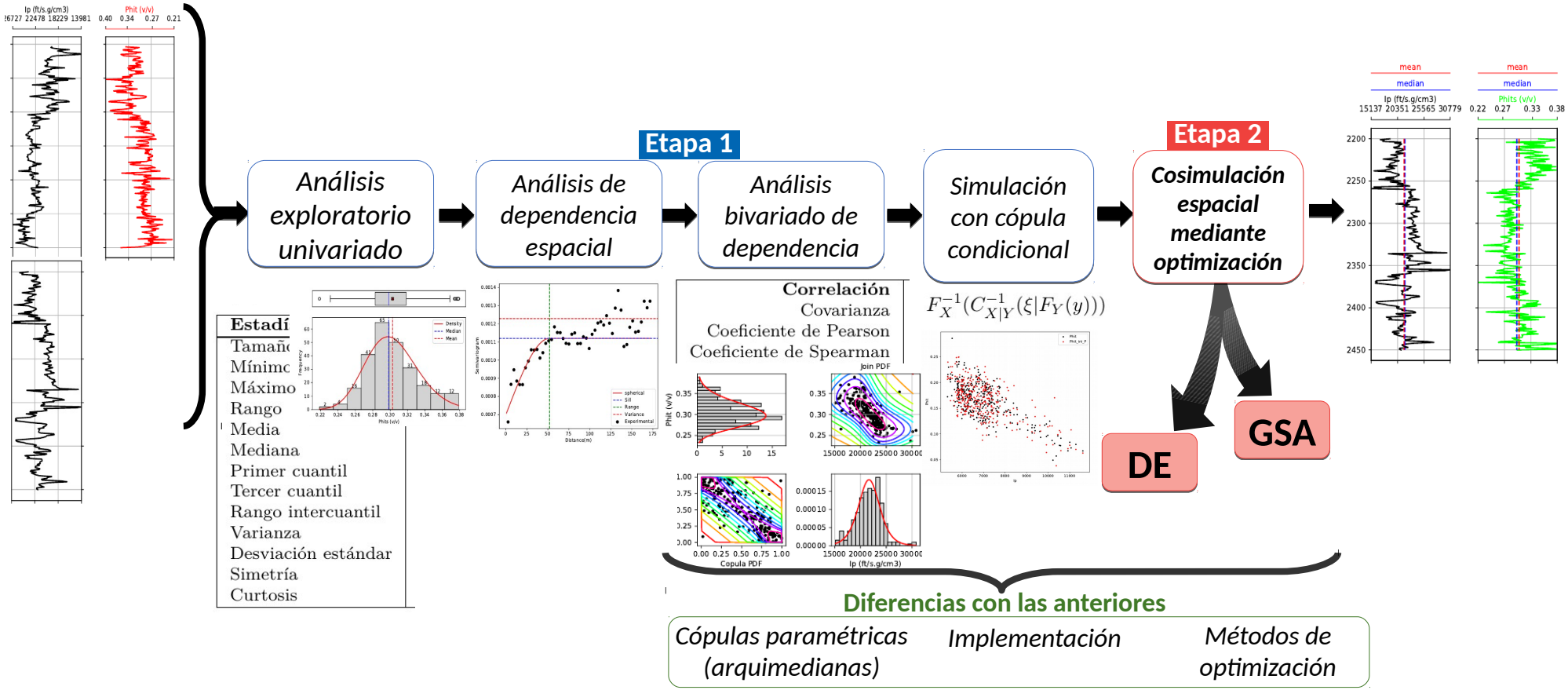



Figura: Metodología de cosimulación estocástica espacial.

# Recocido simulado generalizado

$$g_{q_v}(\Delta x(t)) \propto \frac{[T_{q_v}(t)]^{-\frac{D}{3-q_v}}}{\left[1+(q_v-1)\frac{\Delta x(t)^2}{[T_{q_v}(t)]^{\frac{2}{3-q_v}}}\right]^{\frac{1}{q_v-1}+\frac{D-1}{2}}},$$

$$p_{q_a} = \text{máx}\left\{0, \text{mín}\left\{1, [1-(1-q_a)\,\beta\Delta E]^{\frac{1}{1-4a}}\right\}\right\},$$

Los parámetros de GSA son:

1. temperatura inicial[1] $T_0 = 15$,
2. coeficientes de aceptación[2] $q_a = -5$,
3. coeficientes de visita[3] $q_v = 2{,}62$,
4. razón de reinicio[4] $r = 2e - 5$.

[1] Dréo y col. 2006; Maldonado 2014.
[2] Tsallis y Stariolo 1996.
[3] Tsallis y Stariolo 1996.
[4] Ingber 2000.

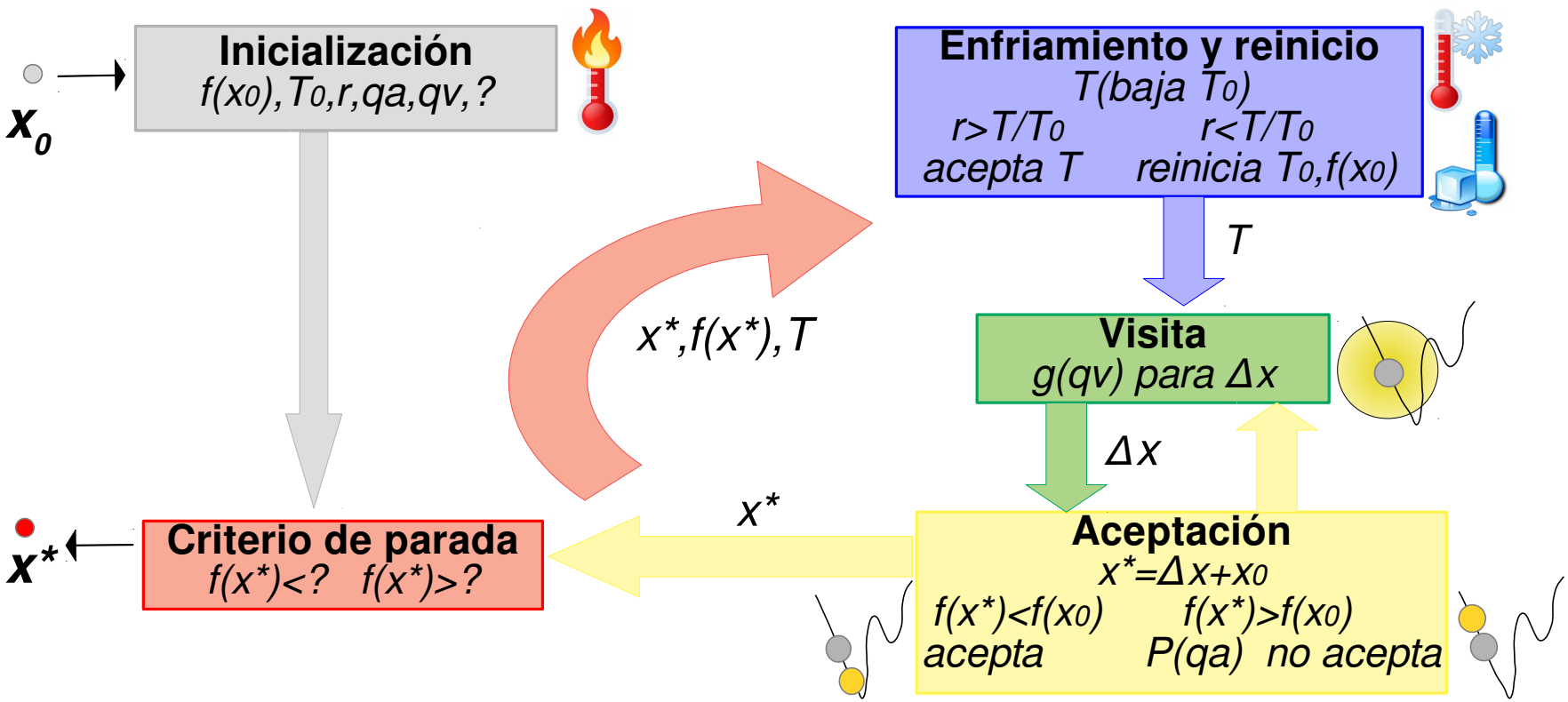



Figura: Esquema del algoritmo GSA.

# Evolución Diferencial

$$v_p = x_c + F \cdot (x_a - x_b)$$

Los parámetros de DE son:

1. tamaño de la población[5] $Np = 35$,
2. probabilidad de cruce $Cr = 0{,}2$,
3. constante de mutación[6] $F = 0{,}5$,
4. estrategia $(s/n/t) = best/1/bin$:
   - $s$ es la forma de seleccionar el vector a perturbar (*best*,*rand* u otros),
   - $n$ es el número de vectores diferencia considerados para la perturbación ($1$ o $2$),
   - $t$ se entiende como el tipo de cruzamiento a ser utilizado (*exp* o *bin*).

[5] Price, Storn y Lampinen 2006.

[6] Price, Storn y Lampinen 2006; Zaharie 2002.

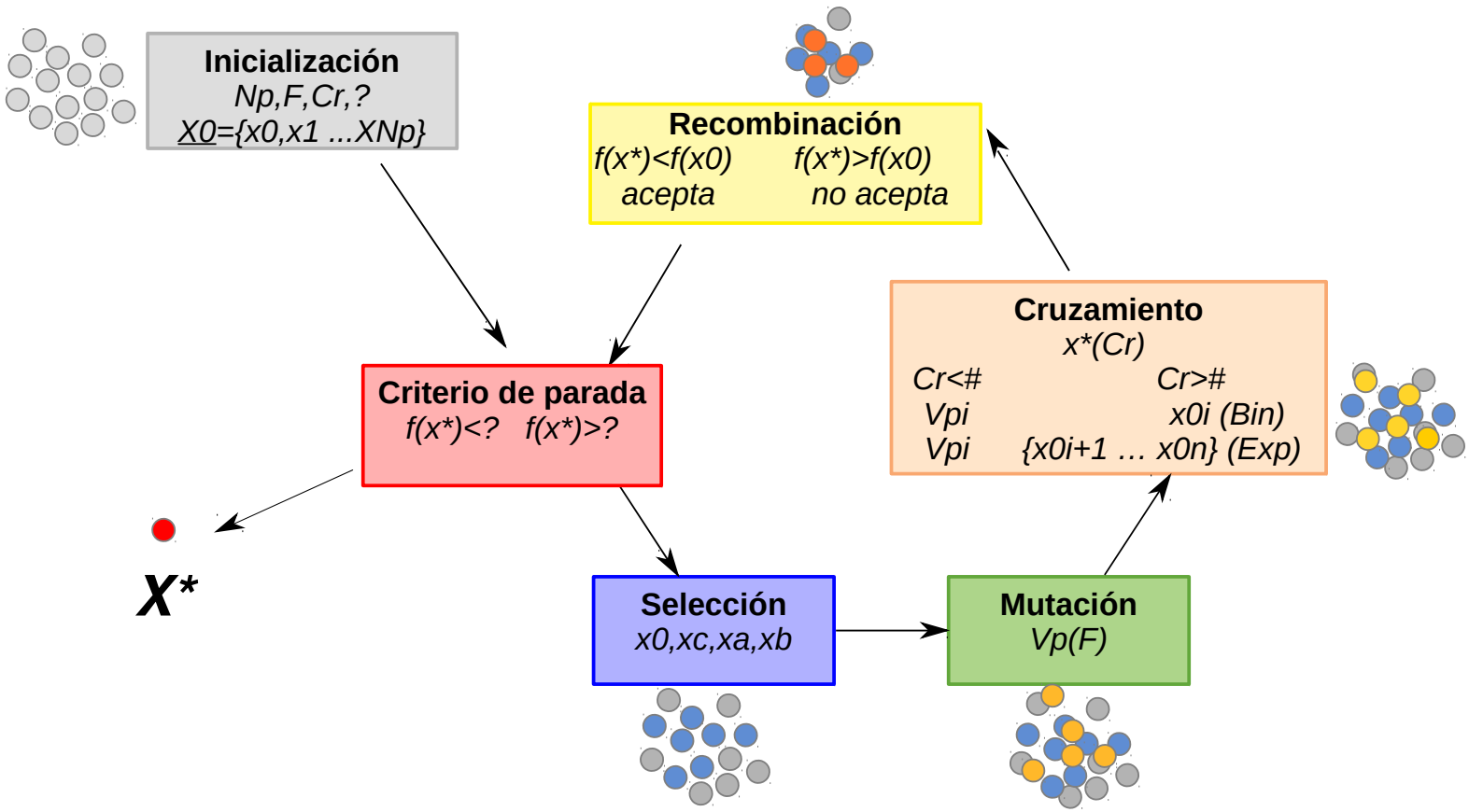



Figura: Esquema del algoritmo DE.

# Problema de optimización

La FMO a optimizar es:

$$\mathcal{O} = \omega_1 \left( \sum_h \frac{[\gamma^*(h) - \gamma(h)]^2}{\gamma(h)^2} \right) + \omega_2 \left( \sum_{x \in \eta_p} \sum_{y \in \eta_s} [F^*(x|y) - F(x|y)]^2 \right)$$

Donde $\gamma$ son los semivariogramas, $F(x|y)$ las distribución condicionales de la variable primaria, $\eta_s$ y $\eta_p$ son las clases secundaria y primaria respectivamente y el asterisco denota la realización simulada.

## Implementación GSA (SciPy)[7]

- La función $dual\_annealing(func, bounds, args = (), parameters*)$ llama a cuatro objetos que se encargan de la optimnización.
- El objeto $ObjectiveFunWrapper$ evalúa la función objetivo y lleva un contador de las veces que se ha evaluado.
- El objeto $EnergyState$ gestiona la información del proceso de optimización.
  $\_init\_$ guarda los parámetros dentro del objeto *(ebest, xbest, current energy, current location y el callback)*.
  $update\_best$ actualiza la posición y valor del mejor, además llama al *callback*.
  $update\_current$ actualiza la posición actual y su valor.
  $reset$ crea $x_0$ si no se entró y lo asigna a la posición actual, calcula el valor de la función para esa posición, luego si no hay una mejor posición la actualiza al igual que su función objetivo.

[7] Virtanen y col. 2020.

## Implementación GSA

- $VisitingDistribution$ genera la posición siguiente a través de la función de distribución de Cauchy-Lorentz ($g_{q_v}$).
  $\_init\_$ guarda el $q_v$ y calcula los factores de la distribución que no dependen de la temperatura ($T$).
  $visit\_fn$ calcula los factores que sí dependen de $T$ y genera $\Delta x$ con la distribución $g_{q_v}$ usando los factores calculados.
  $visiting$ llama a $visit\_fn$ para calcular la nueva posición $x + \Delta x$.
- $StrategyChain$ implementa el criterio de Metropolis.
  $\_init\_$ guarda el mejor valor, la posición y valor actual
  $accept\_reject$ calcula la probabilidad de aceptación y genera un número aleatorio, si es menor se acepta la posición siguiente.
  $run$ genera la posición siguiente utilizado el objeto anterior, si su función objetivo es menor que la actual, actualiza su valor. Si no mejora la función objetivo entonce ejecuta la función anterior.

## Implementación GSA

- $dual\_annealing$: analiza que los datos de entrada estén bien, crea los objetos e inicializa la posición y su valor, luego realiza la optimización. Para esto empieza un ciclo donde primero baja la temperatura del sistema, si es menor que la de reinicio, reinicia la posición y la energía actual, luego optimiza con $strategychain.run$ para la temperatura calculada, si se le indicó búsqueda local se realiza. En este ciclo lleva un conteo de las iteraciones cada vez que disminuye la temperatura. Por último se actualiza el $optimize\_result$ y se retorna.

## Implementación DE (SciPy)[8]

- La función $differential\_evolution(func, bounds, args = (), parameters*)$ llama al objeto $DifferentialEvolutionSolver$. Este objeto tiene varias funciones.
- $\_init\_$, $initpopulationarray$ .
- $x$, $calculate\_population\_energies$ y $promote\_lowest\_energy$
- La función $mutate$ realiza la mutación y cruzamiento de acuerdo a la estrategia especificada. Crea el vector $v_p$ con las funciones $select\_samples$ y $best1$, luego hace el cruzamiento correspondiente (binomial o exponencial), creando de esta manera el vector de comparación ($v_c$). La función $accept\_trial$ realiza la selección, aceptando $v_c$ si su valor de función objetivo es menor que la de su padre.

[8] Virtanen y col. 2020.

## Implementación DE

- Se utiliza $next$ para generar la población siguientes. Primero calcula todos los $v_c$, luego los valores de las funciones objetivos y guarda la localización de los $v_c$ de menor valor (los que se van a aceptar), actualiza la población cambiando los $v_c$ en la localización guardada así como los valores de la función objetivo de la misma manera y cambia a primera posición el de menor valor.
- El $solve$ contiene a todas las demás funciones, se encarga de realizar la optimización. La misma la realiza utilizando $next$ para pasar de una generación a la siguiente y en cada generación evalúa las condiciones de parada como el $callback$. Al final guarda los resultados en $optimize\_result(x, fun, ...)$. Si se define $polish$ se realiza una segunda optimización con un algoritmo de búsqueda local y actualiza los valores. Al final retorna el $optimize\_result$.

El programa está compuesto de seis scripts principales, dos por cada método de optimización:

1. $DE\_Validation$ y $GSA\_Validation$ para la validación donde se conoce la variable a simular y se analiza el error.
2. $DE\_Conditional$ y $GSA\_Conditional$ son otros casos de validación pero donde se tienen datos de condicionamiento.
3. $DE\_Aplication$ y $GSA\_Aplication$ destinado a la aplicación donde no se conoce la variable a simular y se construye un modelo a partir de datos de referencia.

Cada uno funcionan de la siguiente manera. En una primera etapa se cargan los datos, se definen los parámetros del variograma y la distribución bivariada y, además, se elimina la pendiente en caso de ser necesario. Por último se generará la función objetivo que será utilizada en los algoritmos de optimización. En este punto del proceso es necesario elegir el método de optimización. Una vez ejecutado se procede a mostrar los resultados y guardarlos.

Además se compone de siete módulos auxiliares:

1. $LoadSave\_data$ para cargar y salvar los datos.
2. $model$ y $variograms$ son adaptaciones de la biblioteca `geostatsmodels`[9], para el cálculo del variograma.
3. $ot\_copula\_conditional\_YE$ dedicado al análisis bivariado de dependencia y al algoritmo de simulación con cópula condicional usando la biblioteca `OpenTURNS`[10].
4. $funcionesO$ es donde se encuentra implementada la función multiobjetivo.
5. En $Grafics$ se encuentran un grupo de funciones utilizadas para distintas gráficas en el análisis estadístico de los resultados.
6. $dual\_annealing$ está dentro de los módulos ya que se le hicieron modificaciones para que fuera compatible con el recocido simulado clásico, por lo tanto no se puede utilizar directamente el correspondiente a la biblioteca SciPy.

[9] Johnson 2019.

[10] Baudin y col. 2015.

# ¡¡¡Muchas gracias!!!

Para cualquier duda contactar al correo yarilisgm@gmail.com

# Referencias bibliográficas


- Baudin, Michaël y col. (2015). "OpenTURNS: An industrial software for uncertainty quantification in simulation". En: *Handbook of Uncertainty Quantification*. Springer-Verlag. Cap. 58, págs. 2001-2038.
- Dréo, Johann y col. (2006). *Metaheuristics for hard optimization*. Springer-Verlag.
- Ingber, Lester (2000). "Adaptive simulated annealing (ASA): Lessons learned". En: *CoRR* cs.MS/0001018.
- Johnson, Connor (2019). *Geostatsmodels*.
- Maldonado, Víctor Miguel Hernández (2014). "Simulación estocástica espacial de propiedades petrofísicas usando cópulas de Bernstein". Inf. téc. Instituto Mexicano del Petróleo.
- Price, Kenneth, Rainer M Storn y Jouni A Lampinen (2006). *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media.

# Referencias bibliográficas


- Tsallis, Constantino y Daniel A Stariolo (1996). "Generalized simulated annealing". En: *Physica A: Statistical Mechanics and its Applications* 233.1-2, págs. 395-406.
- Virtanen, Pauli y col. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". En: *Nature Methods* 17, págs. 261-272.
- Zaharie, Daniela (2002). "Critical values for the control parameters of differential evolution algorithms". En: *Proc. of MENDEL 2002, 8th Int. Conf. on Soft Computing*, págs. 62-67.