Algoritmos Evolutivos

> **Carrera de Especialización en Inteligencia Artificial, Facultad de Ingeniería**
>
> **Universidad de Buenos Aires, Septiembre de 2024**
>
> Edgar David Guarin Castro (davidg@marketpsychdata.com)
>
> Federico Otero (fede.e.otero@gmail.com)

TP 2: Optimización por Enjambre de Partículas

Importando librerías

In [ ]:
```
!pip install pyswarm
```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go

from pyswarm import pso
```

Ejercicio 1

Escribir un algoritmo PSO para la maximización de la función:

$$y = \sin(x) + \sin(x^2)$$

en el intervalo de 0 ≤ x ≤ 10. Ejecutar el algoritmo en Python con los siguientes parámetros: número de partículas = 2, máximo número de iteraciones = 30, coeficientes de aceleración c1 = c2 = 1.49, peso de inercia w = 0.5. De acuerdo a los requisitos anteriores realizar las siguientes consignas:

a. Transcribir en el informe la solución óptima encontrada (dominio) y el valor óptimo (imagen).

Solución:

Antes de escribir el algoritmo PSO, es necesario definir la función objetivo como se muestra a continuación:

In [2]:
```
# funcion objetivo
def funcion_objetivo(x):
    return np.sin(x) + np.sin(x**2)
```

Se definen también los parámetros para ejecutar el algoritmo PSO:

In [3]:
```
# Parámetros
num_particulas = 2  # número de partículas
dim = 1  # dimensiones
cantidad_iteraciones = 30  # máximo número de iteraciones
c1 = 1.49  # componente cognitivo
c2 = 1.49  # componente social
w = 0.5    # factor de inercia
limite_inf = 0  # límite inferior de búsqueda
limite_sup = 10  # límite superior de búsqueda
```

Debido a que la solución encontrada por el algoritmo PSO depende de la posición inicial de las partículas, es necesario ejecutar el algoritmo varias veces hasta encontrar la mejor solución. Por este motivo, se define el algoritmo PSO para la maximización de la función objetivo como la función algo_pso_max que permitirá ejecutarlo varias veces de forma más cómoda:

In [4]:
```python
# Función para ejecutar el PSO una vez y devolver los valores óptimos
def algo_pso_max():
    # Inicialización
    particulas = np.random.uniform(limite_inf, limite_sup, (num_particulas, dim))  # posiciones iniciales de las pa
    velocidades = np.zeros((num_particulas, dim))  # inicialización de la matriz de velocidades en cero

    # Inicialización de pbest y gbest
    pbest = particulas.copy()  # mejores posiciones personales iniciales
    fitness_pbest = np.array([funcion_objetivo(particulas[i][0]) for i in range(num_particulas)])
    gbest = pbest[np.argmax(fitness_pbest)]  # mejor posición global inicial para maximizar
    fitness_gbest = np.max(fitness_pbest)  # fitness global inicial

    # Almacenamiento para gráficos
    historico_gbest = []

    # Búsqueda
    iteraciones = range(cantidad_iteraciones)
    for iteracion in iteraciones:
        for i in range(num_particulas):  # iteración sobre cada partícula
            r1, r2 = np.random.rand(), np.random.rand()  # generación de dos números aleatorios

            # Actualización de la velocidad de la partícula en cada dimensión
            for d in range(dim):
                velocidades[i][d] = (w * velocidades[i][d] +
                                     c1 * r1 * (pbest[i][d] - particulas[i][d]) +
                                     c2 * r2 * (gbest[d] - particulas[i][d]))

            # Actualización de la posición de la partícula
            for d in range(dim):
                particulas[i][d] = particulas[i][d] + velocidades[i][d]
                # Mantenimiento de las partículas dentro de los límites
                particulas[i][d] = np.clip(particulas[i][d], limite_inf, limite_sup)

            fitness = funcion_objetivo(particulas[i][0])  # Evaluación de la función objetivo para la nueva posició

            # Actualización del mejor personal
            if fitness > fitness_pbest[i]:  # maximizar
                fitness_pbest[i] = fitness
                pbest[i] = particulas[i].copy()
```

```
                # Actualización del mejor global
                if fitness > fitness_gbest:  # maximizar
                    fitness_gbest = fitness
                    gbest = particulas[i].copy()

        # Almacenar el mejor global en cada iteración
        historico_gbest.append((gbest[0], fitness_gbest))

    return gbest, fitness_gbest, historico_gbest
```

A continiuación, el algoritmo es ejecutado 10 veces para garantizar que se encuentre el máximo global siempre, evitando maximos locales, muy comunes debido a la periodicidad de la función objetivo.

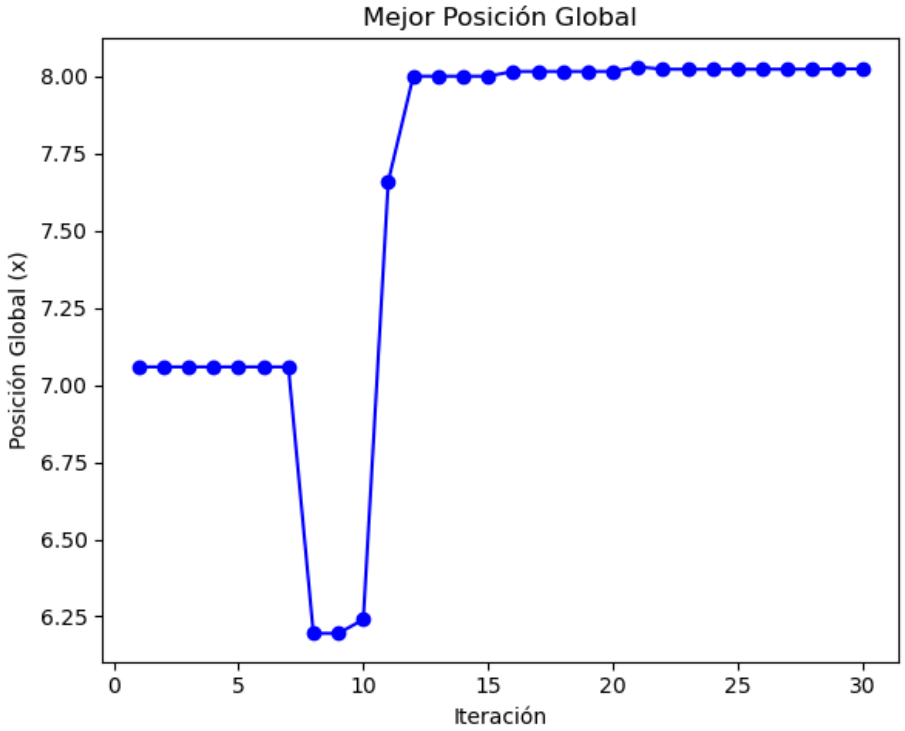
Los mejores resultados son guardados y comparados para obtener los mejores valores de la **solución óptima** correspondiente al valor de x donde la función objetivo es máxima y del **valor óptimo** que equivale al valor máximo de la función objetivo al evaluarla en la solución óptima.

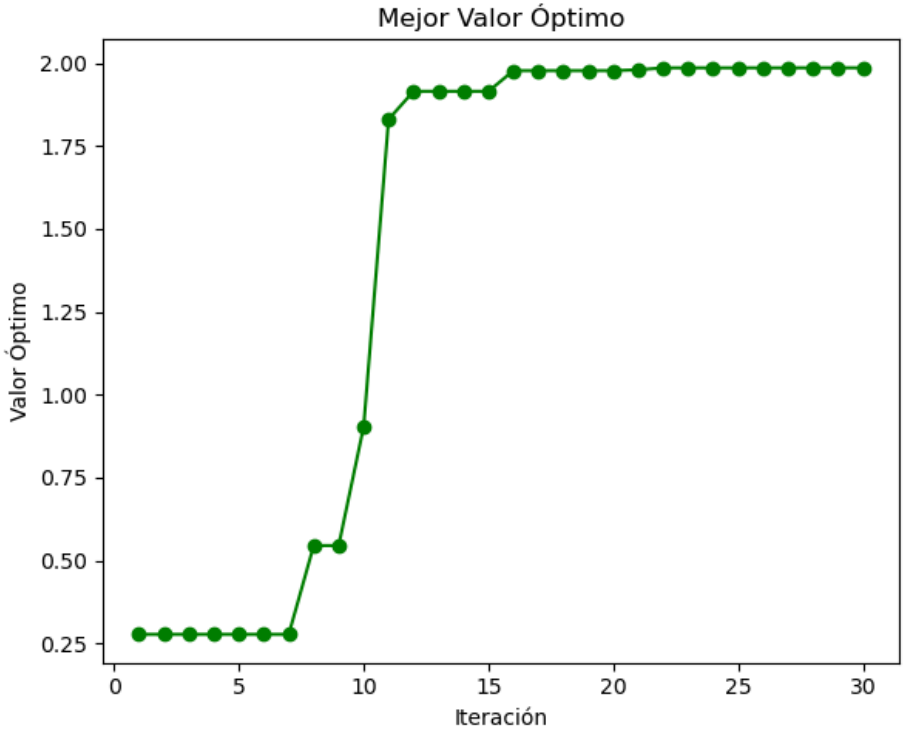
Para facilitar el análisis, se muestran también los gráficos de posición global (azul, izquierda) y de valor óptimo (verde, derecha) en función del número de iteraciones para los mejores resultados obtenidos.

In [8]:
```
# Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
num_rondas = 10
mejores_resultados = []

for _ in range(num_rondas):
    gbest, fitness_gbest, historico_gbest = algo_pso_max()
    mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

# Escoger el mejor resultado de todas las rondas
mejor_ronda = max(mejores_resultados, key=lambda x: x[1])
mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

# Extraer valores para el gráfico
iteraciones = list(range(1, cantidad_iteraciones + 1))
posiciones_gbest = [x[0] for x in mejor_historico_gbest]
valores_optimos = [x[1] for x in mejor_historico_gbest]

# Imprimir la mejor solución
```

```
print("\nMejor solución óptima (x):", mejor_gbest)
print("Mejor valor óptimo:", mejor_fitness_gbest)

# Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
plt.figure(figsize=(12, 5))

# Gráfico de la posición global
plt.subplot(1, 2, 1)
plt.plot(iteraciones, posiciones_gbest, marker='o', linestyle='-', color='b')
plt.title('Mejor Posición Global')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (x)')

# Gráfico del valor óptimo
plt.subplot(1, 2, 2)
plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
plt.title('Mejor Valor Óptimo')
plt.xlabel('Iteración')
plt.ylabel('Valor Óptimo')

plt.tight_layout()
plt.show()
```

```
Mejor solución óptima (x): [8.02454718]
Mejor valor óptimo: 1.9854457589671664
```





Se comprueba que al ejecutar varias veces el algoritmo PSO como se hizo en la celda anterior, se obtiene siempre la solución y el valor óptimo de la función objetivo, aunque la forma de los gráficos anteriores cambie.

En general, se observa que en el gráfico de posición global (azul):

- el algoritmo PSO comienza usualmente explorando el espacio de búsqueda en las primeras iteraciones, con pequeñas variaciones en la posición global óptima encontrada.
- A partir de la iteración 5, hay un cambio más notorio en la posición global, lo que indica que el algoritmo encontró una región con mayor valor de la función objetivo. Aquí el algoritmo empieza a buscar mayores valores de x.
- Si el algoritmo nota que mayores valores de x no maximizan la función, empieza a tomar valores menores de x.
- La mejor posición global se estabiliza luego de 10 iteraciones aproximadamente (proceso de explotación), lo que sugiere que el algoritmo ha convergido hacia un valor cercano al óptimo global. Las fluctuaciones menores después de la iteración 10 pueden deberse a ajustes finos en la búsqueda alrededor de la posición óptima.

- Al final, la solución óptima (x) encontrada por el algoritmo es $x \approx 8.02$ ya que ahí la función objetivo es máxima. Sin embargo, debe tenerse en cuenta que la función puede tener máximos similares en otras posiciones.

Por su parte, el gráfico de valor óptimo (verde) muestra que:

- Al inicio, el valor óptimo no varía como es de esperarse pues la posición global se mantiene igual durante las primeras iteraciones. Sin embargo, después de la quinta iteración, el valor óptimo comienza a maximizarse, mejorando gradualmente. Esto es típico de los algoritmos de optimización basados en partículas, que empiezan explorando de forma más amplia el espacio de soluciones para luego enfocarse en regiones específicas.
- A partir de la iteración 6, hay un incremento más pronunciado en el valor óptimo, lo que indica que las partículas han comenzado a explorar una región más prometedora del espacio.
- Después de la iteración 10, el valor óptimo se estabiliza en su valor máximo, lo que sugiere que el algoritmo ha encontrado una solución que está cerca del óptimo global y ya no realiza mejoras significativas en las iteraciones posteriores.
- En este caso, el valor óptimo (máximo) de la función objetivo $y = \sin(x) + \sin(x^2)$ es $y_{\max}(8.02) \approx 1.99$

b. Indicar la URL del repositorio en donde se encuentra el algoritmo PSO.

Solución:

Este notebook se puede encontrar en el siguiente repositorio: https://github.com/fede0ter0/ceia_algoritmos_evolutivos.git carpeta TP2.

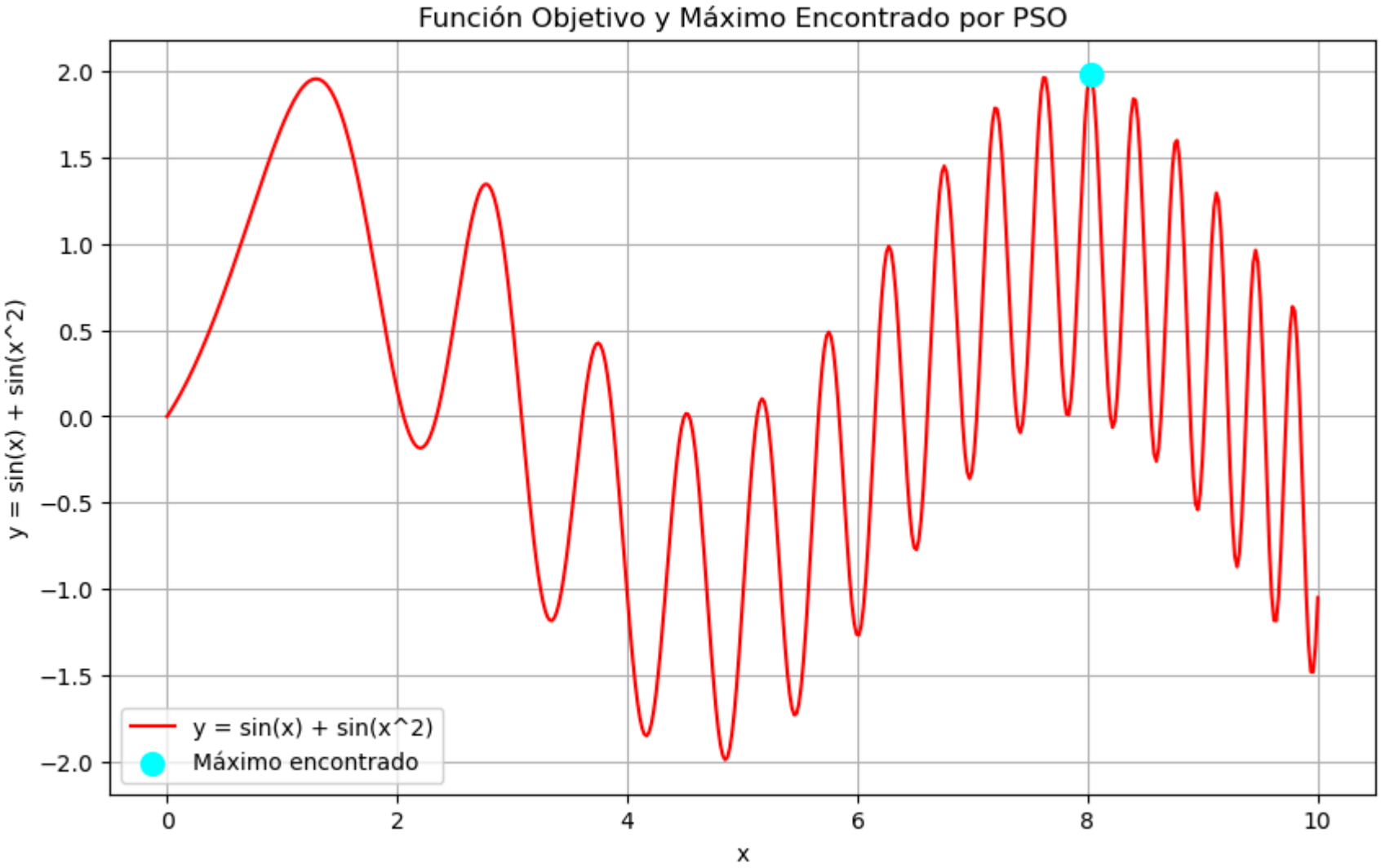
c. Graficar usando matplotlib la función objetivo y agregar un punto celeste en donde el algoritmo haya encontrado el valor máximo. El gráfico debe contener etiquetas en los ejes, leyenda y un título.

Solución:

Para observar mejor la función objetivo y la posición del máximo, se grafica la función a continuación y la posición del valor máximo encontrado anteriormente:

In [9]:
```
# Definir el rango para graficar la función objetivo
x_values = np.linspace(limite_inf, limite_sup, 500)
y_values = funcion_objetivo(x_values)

# Obtener la mejor posición y el valor óptimo encontrado por el algoritmo
mejor_posicion = mejor_gbest[0]
```

```
valor_optimo = mejor_fitness_gbest

# Graficar la función objetivo
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, label='y = sin(x) + sin(x^2)', color='red')
plt.scatter(mejor_posicion, valor_optimo, color='cyan', s=100, label='Máximo encontrado', zorder=5)

# Añadir etiquetas, título y leyenda
plt.xlabel('x')
plt.ylabel('y = sin(x) + sin(x^2)')
plt.title('Función Objetivo y Máximo Encontrado por PSO')
plt.legend()
plt.grid(True)
plt.show()
```



La gráfica muestra la forma oscilatoria de la función objetivo en el intervalo [0,10]. Estas oscilaciones hacen que la función tenga múltiples máximos y mínimos que complican la implementación del algoritmo PSO.

No obstante, debido a que el algoritmo fue ejecutado varias veces, fue posible encontrar uno de los máximos de esa función como se indica con el punto azul celeste. De hecho se observa que el algoritmo logró encontrar uno de los picos más altos de la función dentro del rango considerado. Esto valida la efectividad del PSO en este caso para buscar el valor máximo en una función con múltiples picos.

En resumen, el algoritmo PSO fue capaz de identificar una solución que se encuentra muy cerca del máximo global de la función objetivo, lo cual es un buen resultado dada la naturaleza oscilatoria y compleja de la función.

d. Realizar un gráfico de línea que muestre gbest en función de las iteraciones realizadas.

Solución: Este gráfico ya fue generado y comentado en el punto a. de este ejercicio (gráfico azul).

e. Realizar observaciones/comentarios/conclusiones sobre los resultados obtenidos.

Solución: Las observaciones, comentarios y conclusiones se encuentran en los puntos anteriores.

Ejercicio 2

Dada la siguiente función perteneciente a un paraboloide elíptico de la forma:

$$f(x, y) = (x - a)^2 + (y + b)^2$$

donde, las constantes a y b son valores reales ingresados por el usuario a través de la consola, con intervalos de:

$-100 \leq x \leq 100; x \in \mathbb{R}$

$-100 \leq y \leq 100; y \in \mathbb{R}$

$-50 \leq a \leq 50; a \in \mathbb{R}$

$-50 \leq b \leq 50; b \in \mathbb{R}$

escribir en Python un algoritmo PSO para la minimización de la función paraboloide con los siguientes parámetros: número de partículas = 20, máximo número de iteraciones = 10, coeficientes de aceleración c1 = c2 = 2, peso de inercia w = 0.7, y que cumpla con las siguientes consignas:

a. Transcribir en el informe la solución óptima encontrada (dominio) y el valor óptimo (imagen).

Solución:

Inicialmente se le pide al usuario que introduzca los valores de a y b para la función objetivo:

In [10]:
```
print("Introduzca los valores de a y b (entre -50 y 50) para la función paraboloide:")
a = float(input("a = "))

while a < -50 or a > 50:
    a = float(input("Seleccione un valor entre -50 y 50 \n a = "))

b = float(input("b = "))

while b < -50 or b > 50:
    b = float(input("Seleccione un valor entre -50 y 50 \n b = "))

print("Valores seleccionados:")
print("a = ", a, " b = ", b)
```

```
Introduzca los valores de a y b (entre -50 y 50) para la función paraboloide:
a = 10
b = 30
Valores seleccionados:
a =  10.0  b =  30.0
```

Se define la función objetivo de dos variables a ser minimizada:

In [11]:
```
# funcion objetivo con dos variables
def funcion_objetivo_2D(x,y):
    return (x-a)**2 + (y+b)**2
```

Se definen también los parámetros para ejecutar el algoritmo PSO:

In [12]:
```
# Parámetros
num_particulas = 20  # número de partículas
dim = 2  # dimensiones
cantidad_iteraciones = 10  # máximo número de iteraciones
c1 = 2     # componente cognitivo
c2 = 2     # componente social
w = 0.7    # factor de inercia
limite_inf = -100  # límite inferior de búsqueda
limite_sup = 100   # límite superior de búsqueda
```

Debido a que la solución encontrada por el algoritmo PSO depende de la posición inicial de las partículas, es necesario ejecutar el algoritmo varias veces hasta encontrar la mejor solución. Por este motivo, se define el algoritmo PSO para la minimización de la función objetivo como la función algo_pso_min que permitirá ejecutarlo varias veces de forma más cómoda:

In [13]:
```
# Función para ejecutar el PSO una vez y devolver los valores óptimos
def algo_pso_min():
    # Inicialización
    particulas = np.random.uniform(limite_inf, limite_sup, (num_particulas, dim))  # posiciones iniciales de las pa
    velocidades = np.zeros((num_particulas, dim))  # inicialización de la matriz de velocidades en cero

    # Inicialización de pbest y gbest
    pbest = particulas.copy()  # mejores posiciones personales iniciales
    fitness_pbest = np.array([funcion_objetivo_2D(particulas[i][0], particulas[i][1]) for i in range(num_particulas
    gbest = pbest[np.argmin(fitness_pbest)]  # mejor posición global inicial para maximizar
    fitness_gbest = np.min(fitness_pbest)  # fitness global inicial

    # Almacenamiento para gráficos
    historico_gbest = []

    # Búsqueda
    iteraciones = range(cantidad_iteraciones)
    for iteracion in iteraciones:
        for i in range(num_particulas):  # iteración sobre cada partícula
            r1, r2 = np.random.rand(), np.random.rand()  # generación de dos números aleatorios

            # Actualización de la velocidad de la partícula en cada dimensión
            for d in range(dim):
                velocidades[i][d] = (w * velocidades[i][d] +
                                     c1 * r1 * (pbest[i][d] - particulas[i][d]) +
                                     c2 * r2 * (gbest[d] - particulas[i][d]))

            # Actualización de la posición de la partícula
            for d in range(dim):
                particulas[i][d] = particulas[i][d] + velocidades[i][d]
                # Mantenimiento de las partículas dentro de los límites
                particulas[i][d] = np.clip(particulas[i][d], limite_inf, limite_sup)

            fitness = funcion_objetivo_2D(particulas[i][0], particulas[i][1])  # Evaluación de la función objetivo
```

```
            # Actualización del mejor personal
            if fitness < fitness_pbest[i]:  # maximizar
                fitness_pbest[i] = fitness
                pbest[i] = particulas[i].copy()

                # Actualización del mejor global
                if fitness < fitness_gbest:  # maximizar
                    fitness_gbest = fitness
                    gbest = particulas[i].copy()

        # Almacenar el mejor global en cada iteración
        historico_gbest.append((gbest[0], gbest[1], fitness_gbest))

    return gbest, fitness_gbest, historico_gbest
```

A continiuación, el algoritmo es ejecutado 10 veces para garantizar que se encuentre el mínimo global siempre, evitando mínimos locales.

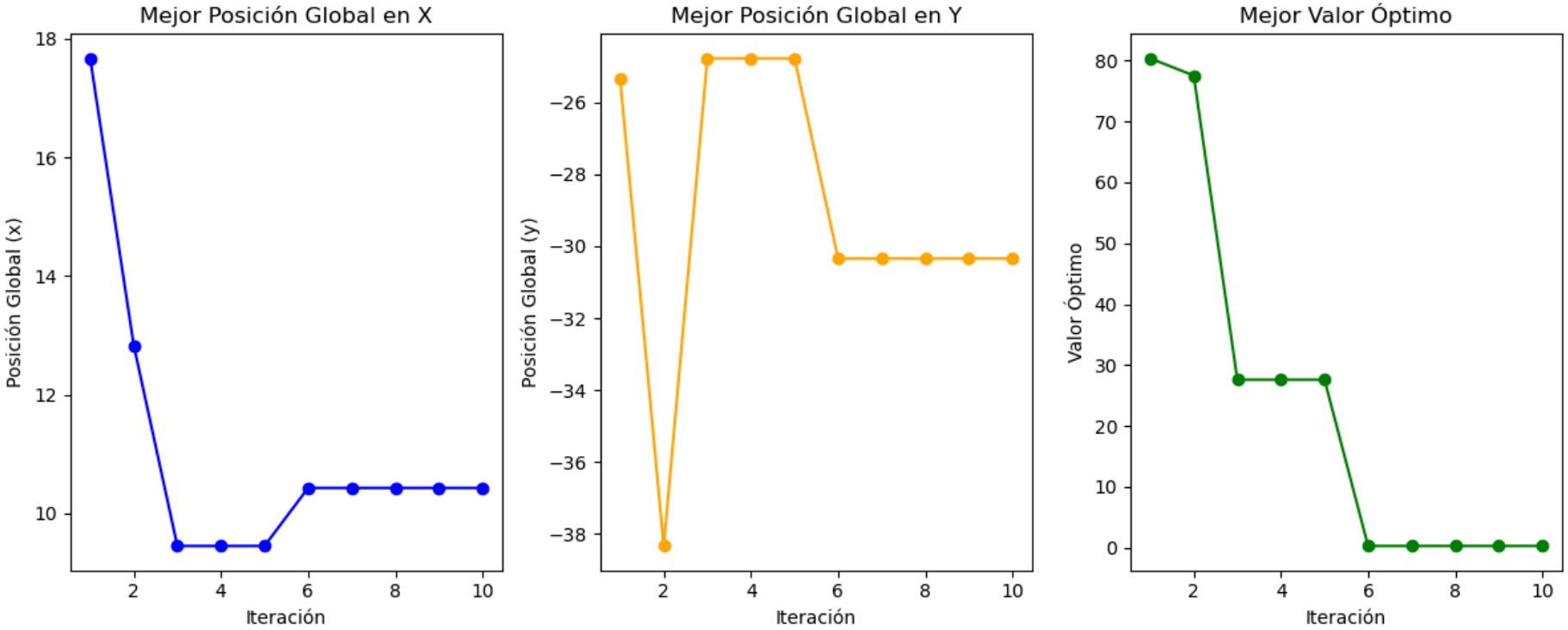
Los mejores resultados son guardados y comparados para obtener los mejores valores de la **solución óptima** correspondiente al valor de x donde la función objetivo es mínima y del **valor óptimo** que equivale al valor mínimo de la función objetivo al evaluarla en la solución óptima.

Para facilitar el análisis, se muestran también los gráficos de posición global (azul, izquierda) y de valor óptimo (verde, derecha) en función del número de iteraciones para los mejores resultados obtenidos.

In [14]:
```
# Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
num_rondas = 10
mejores_resultados = []

for _ in range(num_rondas):
    gbest, fitness_gbest, historico_gbest = algo_pso_min()
    mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

# Escoger el mejor resultado de todas las rondas
mejor_ronda = min(mejores_resultados, key=lambda x: x[1])
mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

# Extraer valores para el gráfico
```

```
iteraciones = list(range(1, cantidad_iteraciones + 1))
posiciones_gbest_x = [x[0] for x in mejor_historico_gbest]
posiciones_gbest_y = [x[1] for x in mejor_historico_gbest]
valores_optimos = [x[2] for x in mejor_historico_gbest]

# Imprimir la mejor solución
print("\nMejor solución óptima (x,y):", mejor_gbest)
print("Mejor valor óptimo:", mejor_fitness_gbest)

# Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
plt.figure(figsize=(12, 5))

# Gráfico de la posición global en x
plt.subplot(1, 3, 1)
plt.plot(iteraciones, posiciones_gbest_x, marker='o', linestyle='-', color='b')
plt.title('Mejor Posición Global en X')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (x)')

# Gráfico de la posición global en y
plt.subplot(1, 3, 2)
plt.plot(iteraciones, posiciones_gbest_y, marker='o', linestyle='-', color='orange')
plt.title('Mejor Posición Global en Y')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (y)')

# Gráfico del valor óptimo
plt.subplot(1, 3, 3)
plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
plt.title('Mejor Valor Óptimo')
plt.xlabel('Iteración')
plt.ylabel('Valor Óptimo')

plt.tight_layout()
plt.show()
```

```
Mejor solución óptima (x,y): [ 10.42583498 -30.34506143]
Mejor valor óptimo: 0.3004028191814466
```



Como en el ejercicio anterior, se comprueba que al ejecutar varias veces el algoritmo PSO, se obtiene siempre la solución y el valor óptimo de la función objetivo, aunque la forma de los gráficos anteriores cambie.

En general, se observa que en el gráfico de posición global en X (azul, izquierda) y en Y (naranja, centro):

- el algoritmo PSO presenta oscilaciones durante la exploración del espacio de búsqueda.
- Usualmente, el algoritmo requiere más de 6 iteraciones para comenzar a estabilizar hacia una posición óptima, lo que sugiere que el algoritmo ha convergido hacia un valor cercano al óptimo global. Las fluctuaciones menores en las últimas iteraciones pueden deberse a ajustes finos en la búsqueda alrededor de la posición óptima.
- Despues de hacer varias pruebas (no presentadas aquí), se observó que la solución óptima se encuentra en el punto $(x, y) \approx (a, -b)$ ya que ahí la función objetivo es mínima.

Por su parte, el gráfico de valor óptimo (verde, derecha) muestra que:

- Al inicio, el valor óptimo puede presentar variaciones pequeñas pero luego decae considerablemente hasta estabilizarse en su valor mínimo, lo que ocurre despues de 6 iteraciones como se mencionó anteriormente.
- Esta disminución pronunciada en el valor óptimo indica que las partículas han comenzado a explorar una región más prometedora del espacio.
- En este caso, el valor óptimo (mínimo) de la función objetivo $f(x, y) = (x - a)^2 + (y + b)^2$ es $f(x, y)_{\min}(a, -b) = 0$, o aproximadamente cero debido a la imprecisión propia del método.

b. Indicar la URL del repositorio en donde se encuentra el algoritmo PSO.

Solución:

Este notebook se puede encontrar en el siguiente repositorio: https://github.com/fede0ter0/ceia_algoritmos_evolutivos.git carpeta TP2.

c. Graficar usando matplotlib la función objetivo f(x, y) en 3D y agregar un punto rojo en donde el algoritmo haya encontrado el valor mínimo. El gráfico debe contener etiquetas en los ejes, leyenda y un título.

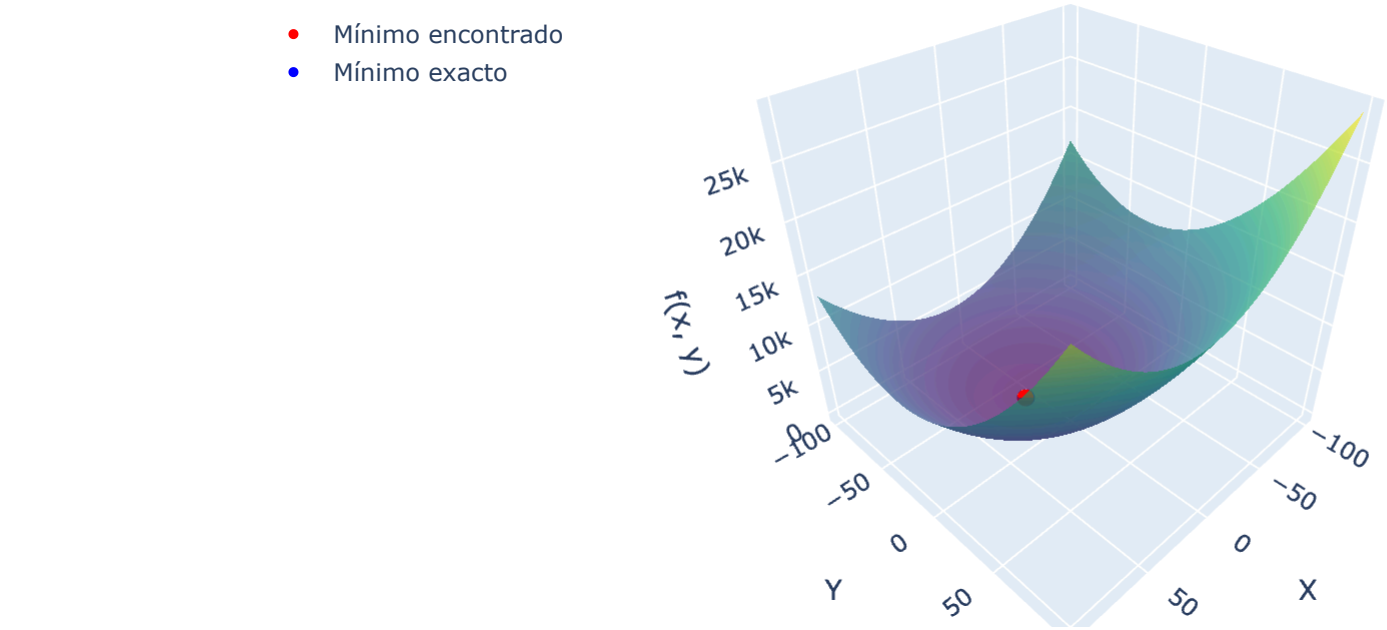
Solución:

Para observar mejor la función objetivo y la posición del mínimo, se representan ambos en el siguiente gráfico interactivo:

In [15]:
```
# Definir el rango para graficar la función
x_vals = np.linspace(limite_inf, limite_sup, 400)
y_vals = np.linspace(limite_inf, limite_sup, 400)
X, Y = np.meshgrid(x_vals, y_vals)

# Evaluar la función para los valores dados de a y b
Z = funcion_objetivo_2D(X, Y)

# Posición global mínima encontrada por PSO (x_min, y_min)
x_min, y_min = mejor_gbest[0], mejor_gbest[1]
z_min = funcion_objetivo_2D(x_min, y_min)

# Crear la superficie con Plotly
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y, colorscale='viridis', opacity=0.7)])

# Agregar el punto rojo donde se encuentra el mínimo
fig.add_trace(go.Scatter3d(
```

```
    x=[x_min],
    y=[y_min],
    z=[z_min],
    mode='markers',
    marker=dict(size=5, color='red'),
    name='Mínimo encontrado'
))

# Agregar el punto azul donde se encuentra el mínimo exacto
fig.add_trace(go.Scatter3d(
    x=[a],
    y=[-b],
    z=[0],
    mode='markers',
    marker=dict(size=5, color='blue'),
    name='Mínimo exacto'
))

# Ajustar etiquetas y título
fig.update_layout(
    title='Paraboloide Elíptico y Mínimo Encontrado',
    scene=dict(
        xaxis_title='X',
        yaxis_title='Y',
        zaxis_title='f(x, y)'
    ),
    legend=dict(x=0.1, y=0.9)
)

# Mostrar el gráfico interactivo
fig.show()

# Diferencias relativas entre las posiciones
x_dif = (x_min - a) * 100 / a
y_dif = (y_min + b) * 100 / -b

print(f"Diferencia relativa entre las posiciones encontrada y esperada en X: {abs(x_dif):.2f}%")
print(f"Diferencia relativa entre las posiciones encontrada y esperada en Y: {abs(y_dif):.2f}%")
```

Paraboloide Elíptico y Mínimo Encontrado




```
Diferencia relativa entre las posiciones encontrada y esperada en X: 4.26%
Diferencia relativa entre las posiciones encontrada y esperada en Y: 1.15%
```

La función sólo posee un mínimo global en el rango considerado. El mínimo exacto se presenta como un punto azul, demostrando que, visualmente, no existe diferencia entre el mínimo encontrado por el método PSO y el valor esperado.

Un cálculo de las diferencias relativas entre las posiciones (x, y) esperada y encontrada muestra que la diferencia es de 1% o menos, lo cual indica la buena precisión del algoritmo PSO para encontrar el mínimo de la función objetivo.

d. Realizar un gráfico de línea que muestre gbest en función de las iteraciones realizadas.

Solución: Este gráfico ya fue presentado y discutido en el apartado a. de este ejercicio (gráfico verde, derecha, y naranja, centro).

e. Establecer el coeficiente de inercia w en 0, ejecutar el algoritmo y realizar observaciones/comentarios/conclusiones sobre los resultados observados.

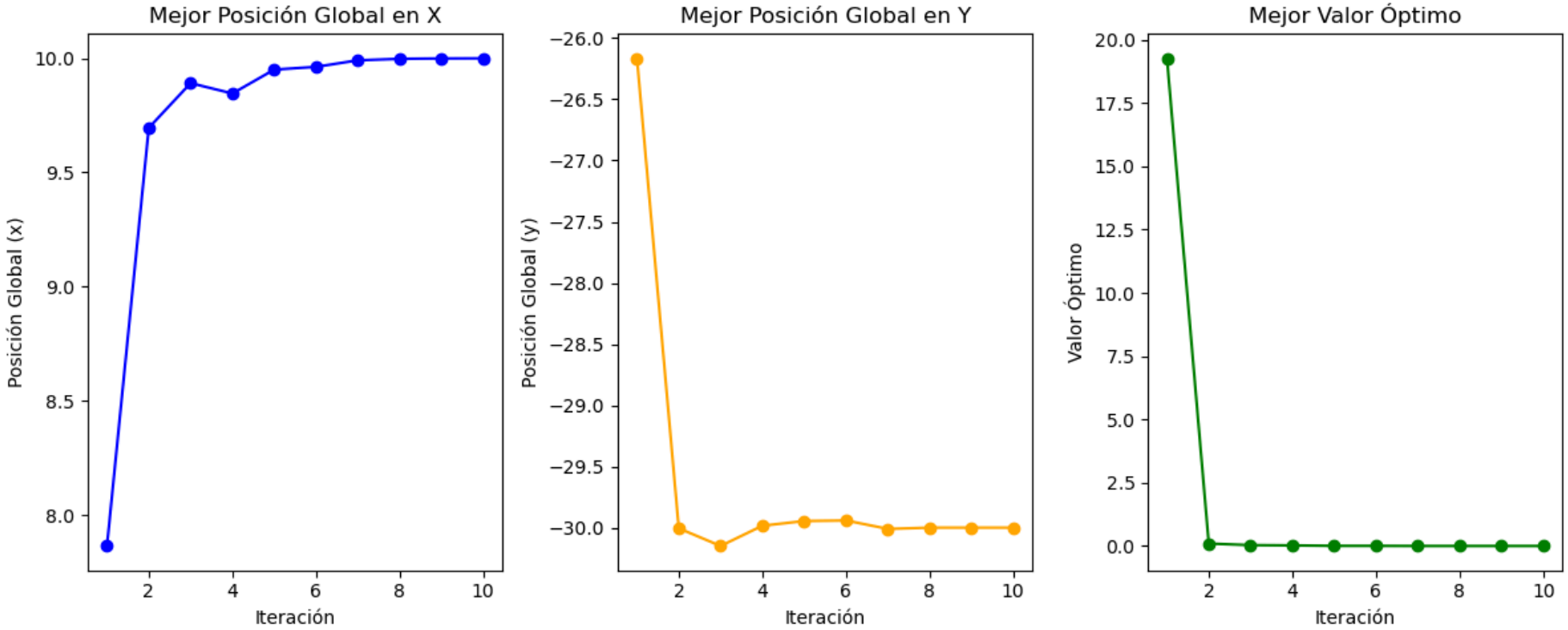
Solución:

El código a continuación muestra los resultados de ejecutar el algoritmo PSO con un peso de inercia nulo:

In [16]:
```
# Anular el peso de inercia
w = 0

# Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
num_rondas = 10
mejores_resultados = []

for _ in range(num_rondas):
    gbest, fitness_gbest, historico_gbest = algo_pso_min()
    mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

# Escoger el mejor resultado de todas las rondas
mejor_ronda = min(mejores_resultados, key=lambda x: x[1])
mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

# Extraer valores para el gráfico
iteraciones = list(range(1, cantidad_iteraciones + 1))
posiciones_gbest_x = [x[0] for x in mejor_historico_gbest]
posiciones_gbest_y = [x[1] for x in mejor_historico_gbest]
valores_optimos = [x[2] for x in mejor_historico_gbest]

# Imprimir la mejor solución
print("\nMejor solución óptima (x,y):", mejor_gbest)
print("Mejor valor óptimo:", mejor_fitness_gbest)
```

```
# Diferencias relativas entre las posiciones
x_dif = (mejor_gbest[0] - a) * 100 / a
y_dif = (mejor_gbest[1] + b) * 100 / -b

print(f"Diferencia relativa entre las posiciones encontrada y esperada en X: {abs(x_dif):.2f}%")
print(f"Diferencia relativa entre las posiciones encontrada y esperada en Y: {abs(y_dif):.2f}%")

# Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
plt.figure(figsize=(12, 5))

# Gráfico de la posición global en x
plt.subplot(1, 3, 1)
plt.plot(iteraciones, posiciones_gbest_x, marker='o', linestyle='-', color='b')
plt.title('Mejor Posición Global en X')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (x)')

# Gráfico de la posición global en y
plt.subplot(1, 3, 2)
plt.plot(iteraciones, posiciones_gbest_y, marker='o', linestyle='-', color='orange')
plt.title('Mejor Posición Global en Y')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (y)')

# Gráfico del valor óptimo
plt.subplot(1, 3, 3)
plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
plt.title('Mejor Valor Óptimo')
plt.xlabel('Iteración')
plt.ylabel('Valor Óptimo')

plt.tight_layout()
plt.show()
```

```
Mejor solución óptima (x,y): [  9.99950229 -29.99927373]
Mejor valor óptimo: 7.751885737547929e-07
Diferencia relativa entre las posiciones encontrada y esperada en X: 0.00%
Diferencia relativa entre las posiciones encontrada y esperada en Y: 0.00%
```



Dado que el peso de inercia w controla el peso dado a la velocidad anterior de las partículas, disminuir o anular su valor hace que se reduzca o se elimine la influencia de la velocidad anterior, favoreciendo la explotación local, es decir, las partículas se mueven más hacia las posiciones ya exploradas, donde se encuentran las mejores posiciones personales y globales.

Lo anterior conlleva a una rápida convergencia como se muestra en los gráficos anteriores de posición global y valor óptimo, en donde la convergencia se alcanza en la mitad de iteraciones en relación con el caso en que $w = 0.7$.

No obstante, anular el peso de inercia genera un mayor riesgo de que las partículas queden atrapadas en un mínimo local, pues se limita la capacidad de las partículas para explorar más allá de las posiciones ya encontradas.

En el caso de la función objetivo analizada aquí, un peso de inercia nulo no representa un riesgo sino una ventaja, ya que en la región considerada sólo existe un mínimo global. Por lo tanto, las partículas siempre van a converger al mismo punto de forma más rápida con una buena precisión como se observa por los valores casi nulos de las diferencias relativas entre las posiciones globales encontradas y las esperadas.

f. Reescribir el algoritmo PSO para que cumpla nuevamente con los ítems A hasta F pero usando la biblioteca pyswarm (from pyswarm import pso).

Solución:

Se implementa a continuación la función PSO de pyswarm y se calcula la diferencia relativa de las posiciones globales encontradas con sus valores esperados:

In [17]:
```
# función objetivo
def funcion_objetivo(x):
    return (x[0]-a)**2 + (x[1]+b)**2


# Parámetros
num_particulas = 20  # número de partículas
cantidad_iteraciones = 10  # máximo número de iteraciones
c1 = 2     # componente cognitivo
c2 = 2     # componente social
w = 0.7    # factor de inercia

lb = [-100, -100]  # limite inf
ub = [100, 100]  # limite sup

# Llamada a la función pso con c1, c2 y w
solucion_optima, valor_optimo = pso(
    funcion_objetivo,
    lb,
    ub,
    swarmsize=num_particulas,
    maxiter=cantidad_iteraciones,
    omega=w,
    phip=c1,
    phig=c2,
    debug=True
    )


# Resultados finales
print("\nSolución óptima (x, y):", solucion_optima)
```

```
print("Valor óptimo:", valor_optimo)

# Diferencias relativas entre las posiciones
x_dif = (solucion_optima[0] - a) * 100 / a
y_dif = (solucion_optima[1] + b) * 100 / -b

print(f"Diferencia relativa entre las posiciones encontrada y esperada en X: {abs(x_dif):.2f}%")
print(f"Diferencia relativa entre las posiciones encontrada y esperada en Y: {abs(y_dif):.2f}%")
```

```
No constraints given.
New best for swarm at iteration 1: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 1: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 2: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 3: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 4: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 5: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 6: [ 15.17783866 -25.49377675] 47.116061161173874
Best after iteration 7: [ 15.17783866 -25.49377675] 47.116061161173874
New best for swarm at iteration 8: [ 15.68112102 -32.10145707] 36.691257821456894
Best after iteration 8: [ 15.68112102 -32.10145707] 36.691257821456894
Best after iteration 9: [ 15.68112102 -32.10145707] 36.691257821456894
Best after iteration 10: [ 15.68112102 -32.10145707] 36.691257821456894
Stopping search: maximum iterations reached --> 10

Solución óptima (x, y): [ 15.68112102 -32.10145707]
Valor óptimo: 36.691257821456894
Diferencia relativa entre las posiciones encontrada y esperada en X: 56.81%
Diferencia relativa entre las posiciones encontrada y esperada en Y: 7.00%
```

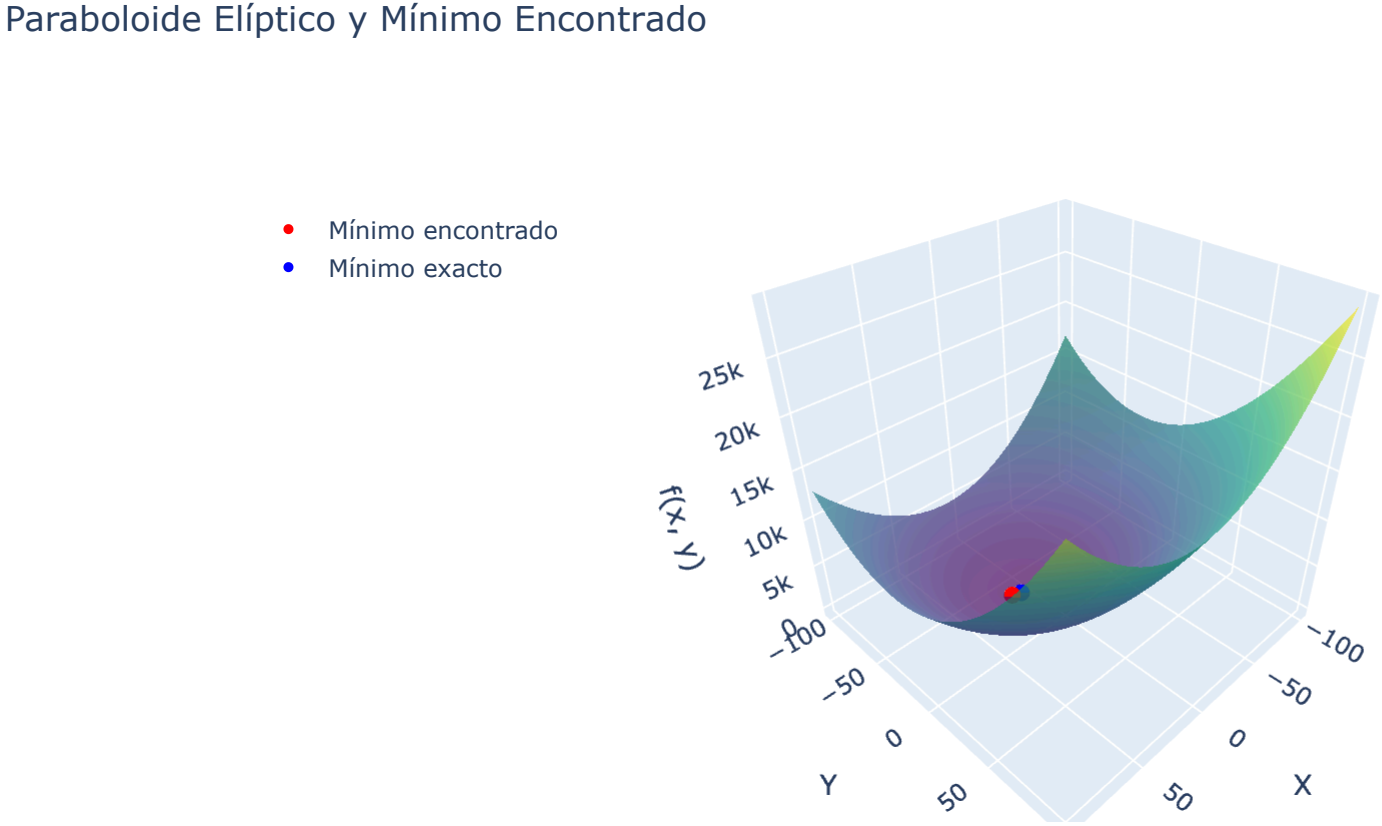
Aunque la función de pyswarm permite una implementación más sencilla del algoritmo PSO, se ve que también tiene el problema de caer en mínimos cercanos al mínimo global. En el ejemplo mostrado anteriormente, las diferencias relativas con los valores esperados son más altas que en los enunciados anteriores.

Dicha diferencia se observa también en el gráfico 3D mostrado a continuación:

In [18]:
```
# Definir el rango para graficar la función
x_vals = np.linspace(limite_inf, limite_sup, 400)
y_vals = np.linspace(limite_inf, limite_sup, 400)
X, Y = np.meshgrid(x_vals, y_vals)
```

```
# Evaluar la función para los valores dados de a y b
Z = funcion_objetivo_2D(X, Y)

# Posición global mínima encontrada por PSO (x_min, y_min)
x_min, y_min = solucion_optima[0], solucion_optima[1]
z_min = funcion_objetivo_2D(x_min, y_min)

# Crear la superficie con Plotly
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y, colorscale='viridis', opacity=0.7)])

# Agregar el punto rojo donde se encuentra el mínimo
fig.add_trace(go.Scatter3d(
    x=[x_min],
    y=[y_min],
    z=[z_min],
    mode='markers',
    marker=dict(size=5, color='red'),
    name='Mínimo encontrado'
))

# Agregar el punto azul donde se encuentra el mínimo exacto
fig.add_trace(go.Scatter3d(
    x=[a],
    y=[-b],
    z=[0],
    mode='markers',
    marker=dict(size=5, color='blue'),
    name='Mínimo exacto'
))

# Ajustar etiquetas y título
fig.update_layout(
    title='Paraboloide Elíptico y Mínimo Encontrado',
    scene=dict(
        xaxis_title='X',
        yaxis_title='Y',
        zaxis_title='f(x, y)'
    ),
    legend=dict(x=0.1, y=0.9)
)
```

```
# Mostrar el gráfico interactivo
fig.show()
```



g. Realizar observaciones/comentarios/conclusiones comparando los resultados obtenidos sin pyswarm y con pyswarm.

Solución:

Por lo anterior, el método PSO requiere más iteraciones o ser ejecutado varias veces como se hizo en los enunciados previos, para converger al mínimo global, ya que la naturaleza del método hace que sea común caer en mínimos locales, diferentes al mínimo global.

Ejercicio 3

Maximizar mediante PSO en Python y con parámetros a elección la función:

$$z = e^{-0.1(x^2+y^2)} \cos(x) \sin(y)$$

donde el intervalo de las variables de decisión se encuentra en el rango -50 <= (x, y) <= 50. En base a las especificaciones mencionadas realizar las siguientes consignas:

a. Transcribir en el informe la solución óptima encontrada (dominio) y el valor óptimo (imagen).

Solución:

Se define la función objetivo:

In [19]:
```
# funcion objetivo con dos variables
def funcion_objetivo_2D(x,y):
    return np.exp(-0.1*(x**2 + y**2)) * np.cos(x) * np.sin(y)
```

Se definen los parámetros:

In [20]:
```
# Parámetros
num_particulas = 30  # número de partículas
dim = 2  # dimensiones
cantidad_iteraciones = 20  # máximo número de iteraciones
c1 = 1.5     # componente cognitivo
c2 = 1.5     # componente social
w = 0.5    # factor de inercia
limite_inf = -50  # límite inferior de búsqueda
limite_sup = 50   # límite superior de búsqueda
```

Se define el algoritmo PSO para la maximización de la función objetivo como la función algo_pso_max que permitirá ejecutarlo varias veces de forma más cómoda:

In [21]:
```python
# Función para ejecutar el PSO una vez y devolver los valores óptimos
def algo_pso_max():
    # Inicialización
    particulas = np.random.uniform(limite_inf, limite_sup, (num_particulas, dim))  # posiciones iniciales de las pa
    velocidades = np.zeros((num_particulas, dim))  # inicialización de la matriz de velocidades en cero

    # Inicialización de pbest y gbest
    pbest = particulas.copy()  # mejores posiciones personales iniciales
    fitness_pbest = np.array([funcion_objetivo_2D(particulas[i][0], particulas[i][1]) for i in range(num_particulas
    gbest = pbest[np.argmax(fitness_pbest)]  # mejor posición global inicial para maximizar
    fitness_gbest = np.max(fitness_pbest)  # fitness global inicial

    # Almacenamiento para gráficos
    historico_gbest = []

    # Búsqueda
    iteraciones = range(cantidad_iteraciones)
    for iteracion in iteraciones:
        for i in range(num_particulas):  # iteración sobre cada partícula
            r1, r2 = np.random.rand(), np.random.rand()  # generación de dos números aleatorios

            # Actualización de la velocidad de la partícula en cada dimensión
            for d in range(dim):
                velocidades[i][d] = (w * velocidades[i][d] +
                                     c1 * r1 * (pbest[i][d] - particulas[i][d]) +
                                     c2 * r2 * (gbest[d] - particulas[i][d]))

            # Actualización de la posición de la partícula
            for d in range(dim):
                particulas[i][d] = particulas[i][d] + velocidades[i][d]
                # Mantenimiento de las partículas dentro de los límites
                particulas[i][d] = np.clip(particulas[i][d], limite_inf, limite_sup)

            fitness = funcion_objetivo_2D(particulas[i][0], particulas[i][1])  # Evaluación de la función objetivo

            # Actualización del mejor personal
            if fitness > fitness_pbest[i]:  # maximizar
                fitness_pbest[i] = fitness
                pbest[i] = particulas[i].copy()
```

```
            # Actualización del mejor global
            if fitness > fitness_gbest:  # maximizar
                fitness_gbest = fitness
                gbest = particulas[i].copy()

        # Almacenar el mejor global en cada iteración
        historico_gbest.append((gbest[0], gbest[1], fitness_gbest))

    return gbest, fitness_gbest, historico_gbest
```

A continiuación, el algoritmo es ejecutado 10 veces para garantizar que se encuentre el máximo global siempre, evitando máximos locales.

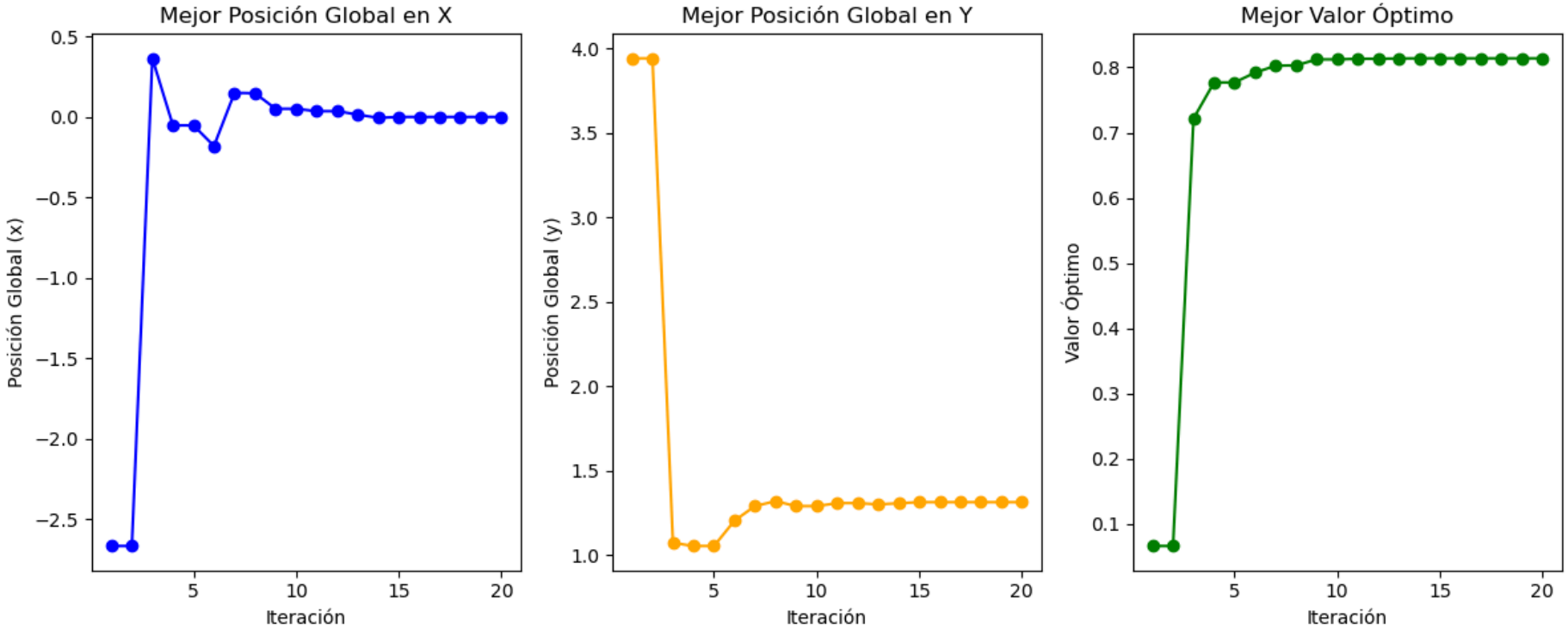
Los mejores resultados son guardados y comparados para obtener los mejores valores de la **solución óptima** correspondiente a la posición (x,y) donde la función objetivo es máxima y del **valor óptimo** que equivale al valor máximo de la función objetivo al evaluarla en la solución óptima.

Para facilitar el análisis, se muestran también los gráficos de posición global en X (azul, izquierda), posición global en Y (naranja, centro) y de valor óptimo (verde, derecha) en función del número de iteraciones para los mejores resultados obtenidos.

In [22]:
```
# Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
num_rondas = 10
mejores_resultados = []

for _ in range(num_rondas):
    gbest, fitness_gbest, historico_gbest = algo_pso_max()
    mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

# Escoger el mejor resultado de todas las rondas
mejor_ronda = max(mejores_resultados, key=lambda x: x[1])
mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

# Extraer valores para el gráfico
iteraciones = list(range(1, cantidad_iteraciones + 1))
posiciones_gbest_x = [x[0] for x in mejor_historico_gbest]
posiciones_gbest_y = [x[1] for x in mejor_historico_gbest]
valores_optimos = [x[2] for x in mejor_historico_gbest]
```

```
# Imprimir la mejor solución
print("\nMejor solución óptima (x,y):", mejor_gbest)
print("Mejor valor óptimo:", mejor_fitness_gbest)

# Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
plt.figure(figsize=(12, 5))

# Gráfico de la posición global en x
plt.subplot(1, 3, 1)
plt.plot(iteraciones, posiciones_gbest_x, marker='o', linestyle='-', color='b')
plt.title('Mejor Posición Global en X')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (x)')

# Gráfico de la posición global en y
plt.subplot(1, 3, 2)
plt.plot(iteraciones, posiciones_gbest_y, marker='o', linestyle='-', color='orange')
plt.title('Mejor Posición Global en Y')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (y)')

# Gráfico del valor óptimo
plt.subplot(1, 3, 3)
plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
plt.title('Mejor Valor Óptimo')
plt.xlabel('Iteración')
plt.ylabel('Valor Óptimo')

plt.tight_layout()
plt.show()
```

```
Mejor solución óptima (x,y): [7.11057018e-04 1.31390954e+00]
Mejor valor óptimo: 0.8138322967823663
```



Los gráficos arriba muestran que, con los parámetros escogidos, se logra una convergencia al máximo global despues de 10 iteraciones, a diferencia de la función del Ejercicio 2, donde la convergencia se obtenía con menos de 5 iteraciones. Esto se debe a que la función objetivo:

$$z = e^{-0.1(x^2+y^2)} \cos(x) \sin(y)$$

presenta una mayor complejidad, al tener oscilaciones pronunciadas cerca al origen, como se verá en el gráfico siguiente.

Despues de hacer varias pruebas (no presentadas aquí), se observó que la solución óptima se encuentra en el punto $(x, y) \approx (0, 1.31)$ ya que ahí la función objetivo es máxima.

Por su parte, el gráfico de valor óptimo (verde, derecha) muestra que:

- Al inicio, el valor óptimo crece considerablemente hasta estabilizarse en su valor máximo.

- Este crecimiento pronunciado en el valor óptimo indica que las partículas han comenzado a explorar una región más prometedora del espacio.
- En este caso, el valor óptimo (máximo) de la función objetivo $f(x, y) \approx 0.81$.

b. Indicar la URL del repositorio en donde se encuentra el algoritmo PSO.

Solución:

Este notebook se puede encontrar en el siguiente repositorio: https://github.com/fede0ter0/ceia_algoritmos_evolutivos.git carpeta TP2.

c. Graficar usando matplotlib la función objetivo z(x, y) y agregar un punto verde en donde el algoritmo haya encontrado el valor máximo. El gráfico debe contener etiquetas en los ejes, leyenda y un título.

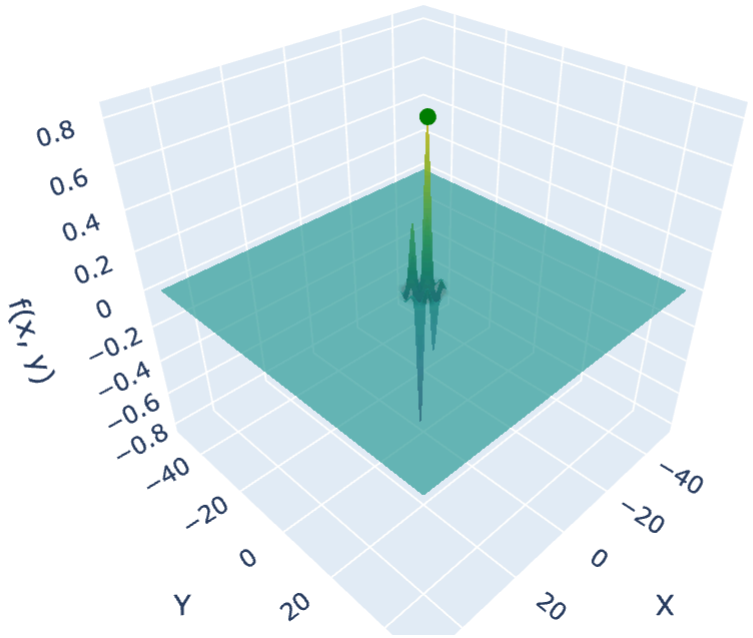
Solución:

Para observar mejor la función objetivo y la posición del máximo, se representan ambos en el siguiente gráfico interactivo:

In [23]:
```
# Definir el rango para graficar la función
x_vals = np.linspace(limite_inf, limite_sup, 400)
y_vals = np.linspace(limite_inf, limite_sup, 400)
X, Y = np.meshgrid(x_vals, y_vals)

# Evaluar la función para los valores dados de a y b
Z = funcion_objetivo_2D(X, Y)

# Posición global mínima encontrada por PSO (x_min, y_min)
x_min, y_min = mejor_gbest[0], mejor_gbest[1]
z_min = funcion_objetivo_2D(x_min, y_min)

# Crear la superficie con Plotly
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y, colorscale='viridis', opacity=0.7)])

# Agregar el punto rojo donde se encuentra el mínimo
fig.add_trace(go.Scatter3d(
    x=[x_min],
    y=[y_min],
    z=[z_min],
    mode='markers',
```

```
    marker=dict(size=5, color='green'),
    name='Mínimo encontrado'
))

# Ajustar etiquetas y título
fig.update_layout(
    title='Paraboloide Elíptico y Mínimo Encontrado',
    scene=dict(
        xaxis_title='X',
        yaxis_title='Y',
        zaxis_title='f(x, y)'
    ),
    legend=dict(x=0.1, y=0.9)
)

# Mostrar el gráfico interactivo
fig.show()
```

Paraboloide Elíptico y Mínimo Encontrado



La función posee un máximo global en la solución óptima (0,1.31) demostrando que el método convergió correctamente.

Esto demuestra su capacidad para lidiar con funciones complejas que involucran términos exponenciales y trigonométricos, responsables de oscilaciones pronunciadas en ciertas regiones del espacio. Además, la forma de la superficie en el gráfico 3D sugiere que el algoritmo ha explotado correctamente las regiones cercanas al óptimo en el espacio de búsqueda

d. Realizar un gráfico de línea que muestre gbest en función de las iteraciones realizadas.

Solución: Este gráfico ya fue presentado y discutido en el apartado a. de este ejercicio (gráfico verde, derecha).

e. Establecer el coeficiente de inercia w en 0, ejecutar el algoritmo y realizar observaciones/comentarios/conclusiones sobre los resultados observados.

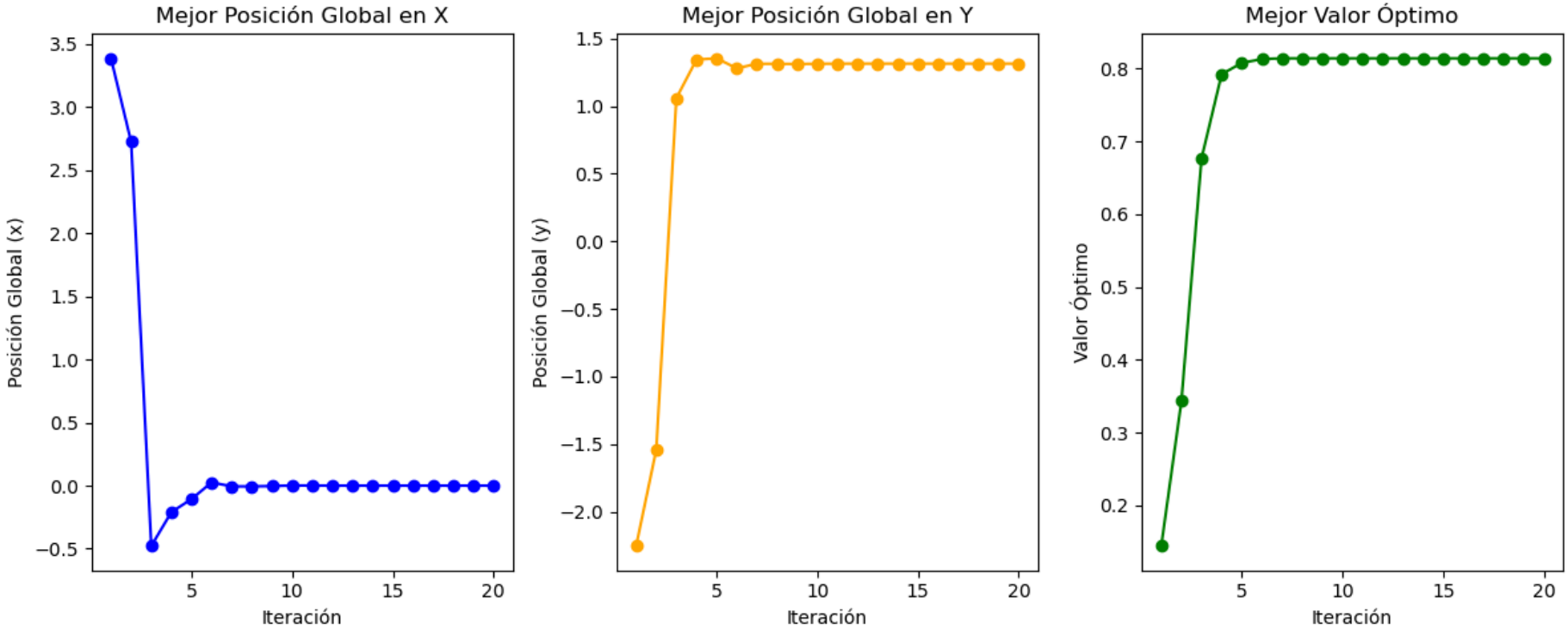
Solución:

En la siguiente celda se ejecuta el algoritmo PSO con $w = 0$:

In [24]:
```
w = 0

# Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
num_rondas = 10
mejores_resultados = []

for _ in range(num_rondas):
    gbest, fitness_gbest, historico_gbest = algo_pso_max()
    mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

# Escoger el mejor resultado de todas las rondas
mejor_ronda = max(mejores_resultados, key=lambda x: x[1])
mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

# Extraer valores para el gráfico
iteraciones = list(range(1, cantidad_iteraciones + 1))
posiciones_gbest_x = [x[0] for x in mejor_historico_gbest]
posiciones_gbest_y = [x[1] for x in mejor_historico_gbest]
valores_optimos = [x[2] for x in mejor_historico_gbest]

# Imprimir la mejor solución
print("\nMejor solución óptima (x,y):", mejor_gbest)
print("Mejor valor óptimo:", mejor_fitness_gbest)

# Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
plt.figure(figsize=(12, 5))

# Gráfico de la posición global en x
```

```
plt.subplot(1, 3, 1)
plt.plot(iteraciones, posiciones_gbest_x, marker='o', linestyle='-', color='b')
plt.title('Mejor Posición Global en X')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (x)')

# Gráfico de la posición global en y
plt.subplot(1, 3, 2)
plt.plot(iteraciones, posiciones_gbest_y, marker='o', linestyle='-', color='orange')
plt.title('Mejor Posición Global en Y')
plt.xlabel('Iteración')
plt.ylabel('Posición Global (y)')

# Gráfico del valor óptimo
plt.subplot(1, 3, 3)
plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
plt.title('Mejor Valor Óptimo')
plt.xlabel('Iteración')
plt.ylabel('Valor Óptimo')

plt.tight_layout()
plt.show()
```

```
Mejor solución óptima (x,y): [-1.33826061e-08  1.31383772e+00]
Mejor valor óptimo: 0.8138325463315369
```



Debido a que la función posee un máximo distintivo y el algoritmo se ejecuta varias veces para garantizar la convergencia al óptimo global, un valor nulo de w no representa un peligro que lleve al algoritmo a caer en máximos globales. Por el contrario, es una ventaja que ayuda al método a converger rápidamente al óptimo global en menos de la mitad de iteraciones empleadas cuando se usó $w = 0.5$.

f. Graficar 3 boxplots usando gbest, uno para w = 0.8, w = 0.5 y el otro para w = 0.

Solución:

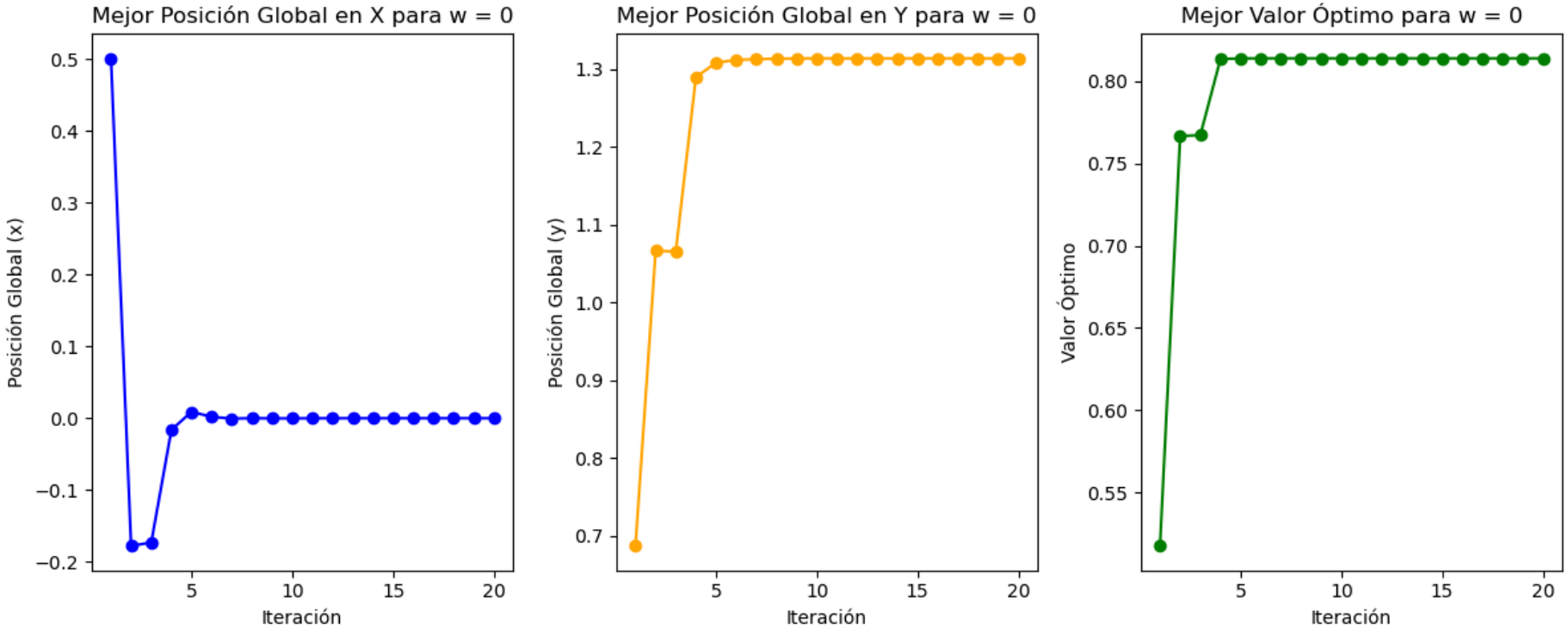
Los gráficos para cada valor de w son presentados a continuación:

In [25]:
```
for w in [0, 0.5, 0.8]:
    # Ejecutar el algoritmo PSO 10 veces y guardar los mejores resultados
    num_rondas = 10
    mejores_resultados = []

    for _ in range(num_rondas):
```

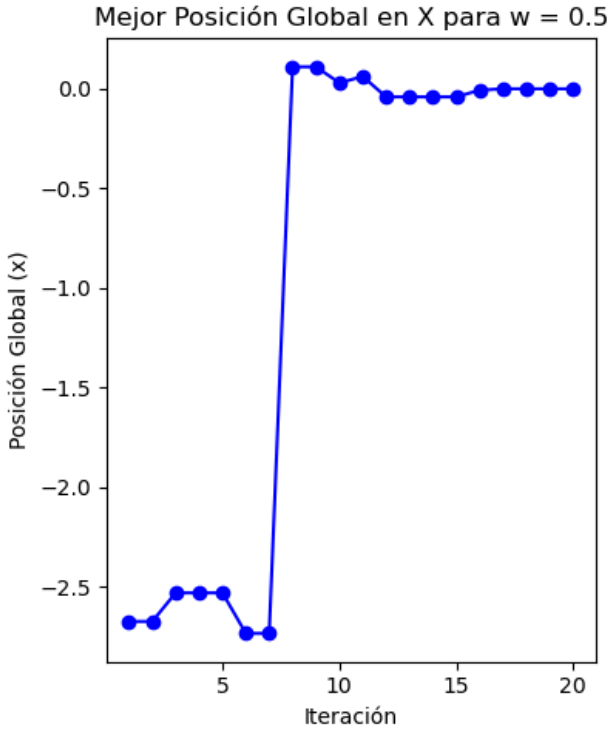
```
        gbest, fitness_gbest, historico_gbest = algo_pso_max()
        mejores_resultados.append((gbest, fitness_gbest, historico_gbest))

    # Escoger el mejor resultado de todas las rondas
    mejor_ronda = max(mejores_resultados, key=lambda x: x[1])
    mejor_gbest, mejor_fitness_gbest, mejor_historico_gbest = mejor_ronda

    # Extraer valores para el gráfico
    iteraciones = list(range(1, cantidad_iteraciones + 1))
    posiciones_gbest_x = [x[0] for x in mejor_historico_gbest]
    posiciones_gbest_y = [x[1] for x in mejor_historico_gbest]
    valores_optimos = [x[2] for x in mejor_historico_gbest]

    # Imprimir la mejor solución
    print("\nResultados para w = ",w)
    print("\nMejor solución óptima (x,y):", mejor_gbest)
    print("Mejor valor óptimo:", mejor_fitness_gbest)

    # Gráfico de la mejor posición global y el valor óptimo en función del número de iteraciones
    plt.figure(figsize=(12, 5))

    # Gráfico de la posición global en x
    plt.subplot(1, 3, 1)
    plt.plot(iteraciones, posiciones_gbest_x, marker='o', linestyle='-', color='b')
    plt.title(f'Mejor Posición Global en X para w = {w}')
    plt.xlabel('Iteración')
    plt.ylabel('Posición Global (x)')

    # Gráfico de la posición global en y
    plt.subplot(1, 3, 2)
    plt.plot(iteraciones, posiciones_gbest_y, marker='o', linestyle='-', color='orange')
    plt.title(f'Mejor Posición Global en Y para w = {w}')
    plt.xlabel('Iteración')
    plt.ylabel('Posición Global (y)')

    # Gráfico del valor óptimo
    plt.subplot(1, 3, 3)
    plt.plot(iteraciones, valores_optimos, marker='o', linestyle='-', color='g')
    plt.title(f'Mejor Valor Óptimo para w = {w}')
    plt.xlabel('Iteración')
    plt.ylabel('Valor Óptimo')
```

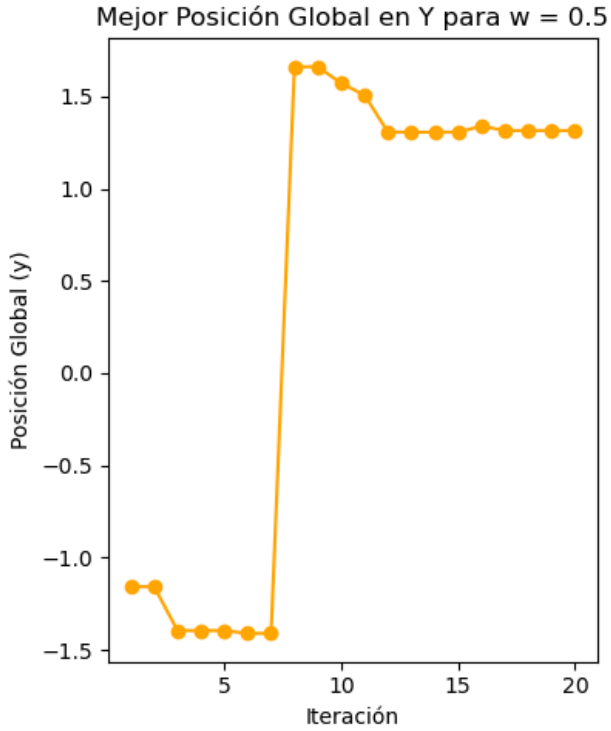
```
    plt.tight_layout()
    plt.show()
```

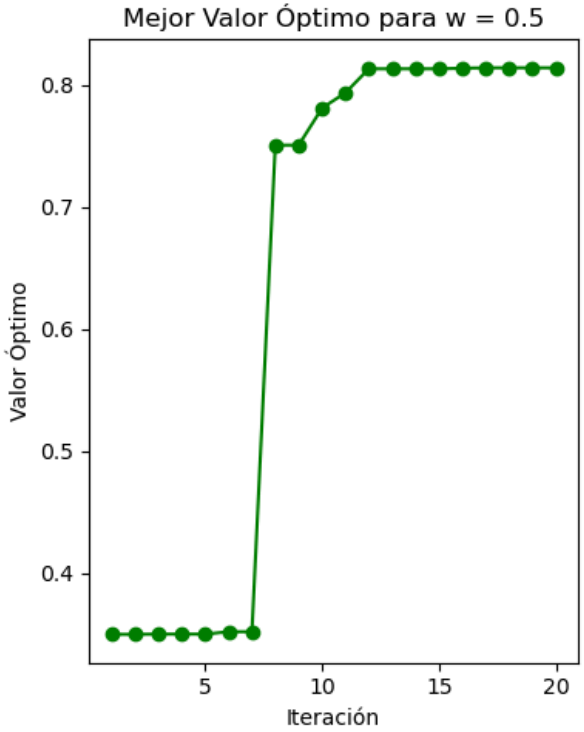
```
Resultados para w =  0

Mejor solución óptima (x,y): [-9.52317452e-09  1.31383769e+00]
Mejor valor óptimo: 0.8138325463315368
```



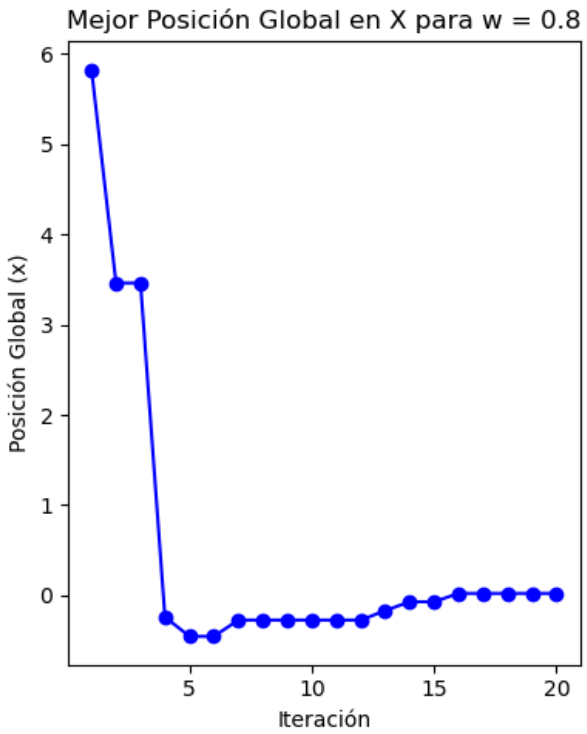

```
Resultados para w =  0.5

Mejor solución óptima (x,y): [7.73929193e-04 1.31491137e+00]
Mejor valor óptimo: 0.8138316586849037
```

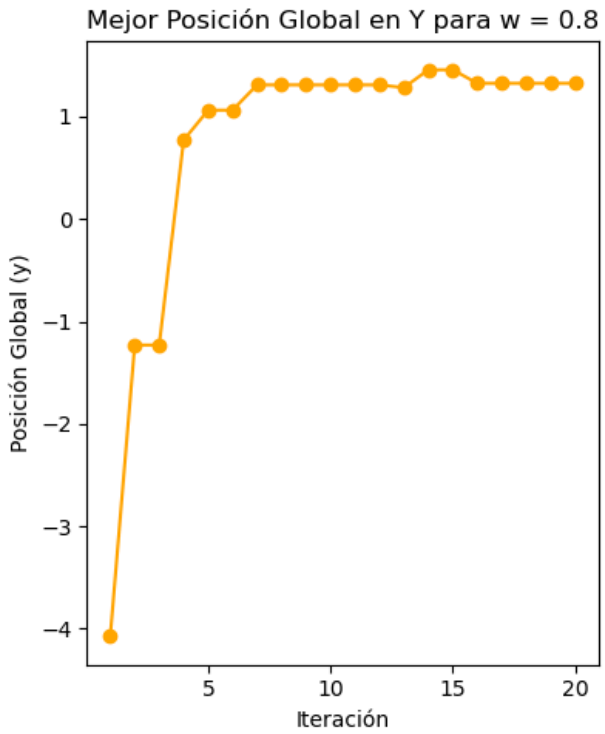


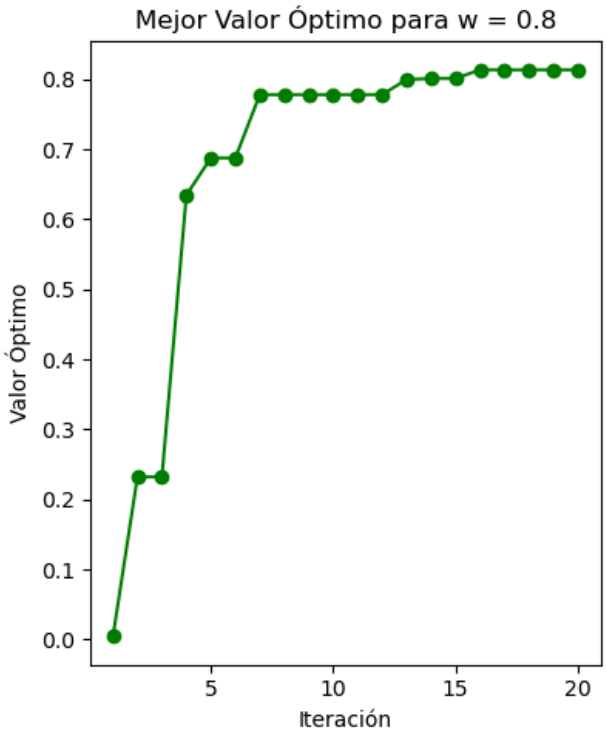





```
Resultados para w =  0.8

Mejor solución óptima (x,y): [0.0227788  1.32548055]
Mejor valor óptimo: 0.813509344895209
```







g. Explicar interpretando la información que proporcionan los 3 boxplots.

Solución:

De los resultados anteriores se infiere que cuando w=0:

- Las posiciones convergen rápidamente hacia la solución óptima en las primeras iteraciones. Después de esa rápida convergencia, las posiciones se mantienen estables sin cambios significativos.
- Hay una rápida mejora en el valor óptimo en las primeras 4 iteraciones.
- Las partículas no conservan ninguna parte de su velocidad anterior. Esto se traduce en una rápida explotación del espacio de búsqueda, lo que permite a las partículas converger rápidamente hacia el mejor valor encontrado. Sin embargo, esta rápida convergencia indica una limitada capacidad de exploración, lo que puede aumentar el riesgo de quedar atrapado en mínimos locales. Este riesgo se reduce al ejecutar varias rondas del algoritmo.

Cuando w=0.5, se observa que:

- Las posiciones globales (x,y) muestran un patrón más gradual en su convergencia. Hay ligeros ajustes y oscilaciones en las posiciones, especialmente en y, lo que indica que las partículas están explorando el espacio de búsqueda de una manera más equilibrada.
- El valor óptimo mejora de manera pronunciada en las primeras iteraciones y luego se estabiliza cerca de 0.8, similar a lo que ocurre con w=0. Sin embargo, las oscilaciones en las posiciones indican que el algoritmo continúa explorando el espacio de búsqueda.
- Hay un equilibrio entre la exploración global y la explotación local. Las partículas retienen parte de su velocidad anterior, permitiendo un ajuste más fino y manteniendo un nivel moderado de exploración, lo cual es favorable para encontrar un óptimo cercano al global.

Cuando w=0.8, se tiene que:

- Las posiciones muestran más variación y oscilaciones a lo largo de las iteraciones. En x, hay un cambio significativo antes de estabilizarse, mientras que en y hay fluctuaciones notables incluso después de la convergencia inicial.
- La mejora en el valor óptimo es más gradual en comparación con los valores de w anteriores. Hay un patrón de ajustes más pequeños y continuos a lo largo de las iteraciones, lo que indica una mayor exploración.
- Las partículas conservan más de su velocidad anterior, lo que permite una exploración más extensa del espacio de búsqueda. Este comportamiento puede ser útil para escapar de mínimos locales y explorar otras regiones. Sin embargo, la convergencia es más lenta y menos estable, lo que puede resultar en un valor óptimo final menos preciso.

En conclusión, un valor nulo de w conduce a una rápida convergencia, mientras que valores más altos favorecen una mayor exploración y una convergencia más lenta. En este caso, los riesgos que implican un w nulo son minimizados al ejecutar el algoritmo varias veces, para garantizar la escogencia de los mejores valores de entre todos los intentos.

Sin embargo, si se desea tener mayor seguridad al obtener los resultados, un valor de w=0.5 parece proporcionar un equilibrio entre exploración y explotación, permitiendo una mejora rápida y, a la vez, una búsqueda adicional para ajustar las posiciones.

Ejercicio 4

Mediante PSO es posible resolver en forma aproximada un sistema de n ecuaciones con n incógnitas clásico del tipo:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases}$$

Por ejemplo, el siguiente es un sistema de 2 ecuaciones con 2 incógnitas (x1 y x2) que puede ser resuelto con PSO:

$$\begin{cases} 3x_1 + 2x_2 = 9 \\ x_1 - 5x_2 = 4 \end{cases}$$

Utilizando la biblioteca pyswarm:

a. Escribir un algoritmo PSO con parámetros a elección (c1, c2, w, número de partículas, máximo número de iteraciones) que encuentre x1 y x2 para el sistema de ecuaciones anterior. Transcribir en el informe el código fuente.

Solución:

El sistema de ecuaciones:

$$\begin{cases} 3x_1 + 2x_2 = 9 \\ x_1 - 5x_2 = 4 \end{cases}$$

se puede solucionar por método de substitución, de modo que de la primera ecuación se tiene que:

$$x_1 = \frac{9}{3} - \frac{2x_2}{3}$$

Substituyendo en la segunda ecuación y solucionando para x_2, se obtiene que $x_2 = -3/13$. Reemplazando este valor en x_1 se llega a que $x_1 = 41/13$. Así, las soluciones exactas son:

$$\begin{cases} x_1 = \frac{41}{13} \approx 3.15 \\ x_2 = -\frac{3}{13} \approx -0.23 \end{cases}$$

Para llegar a dichas soluciones usando el algoritmo PSO, se propone definir la función objetivo como la suma de los errores cuadrados para ambas ecuaciones. Dado que se quieren encontrar los valores de x_1 y x_2 que satisfacen el sistema, la función objetivo se

construirá para penalizar las desviaciones de las ecuaciones. Por lo tanto, la función objetivo será:

$$f(x) = (3x_1 + 2x_2 - 9)^2 + (x_1 - 5x_2 - 4)^2$$

con $x = [x_1, x_2]$, de modo que:

In [26]:
```
# Definir la función objetivo
def funcion_objetivo(x):
    x1, x2 = x[0], x[1]
    eq1 = 3 * x1 + 2 * x2 - 9
    eq2 = x1 - 5 * x2 - 4
    # Suma de los errores al cuadrado
    return eq1**2 + eq2**2
```

Se definen los parámetros del algoritmo, los cuales fueron escogidos ya que generaron los mejores resultados en una serie de pruebas no presentadas aquí:

In [27]:
```
# Parámetros del PSO
num_particulas = 40  # Número de partículas
cantidad_iteraciones = 50  # Máximo número de iteraciones
c1 = 2     # Componente cognitiva
c2 = 2     # Componente social
w = 0      # Factor de inercia

# Límites de búsqueda
lb = [-5, -5]  # Límite inferior
ub = [5, 5]    # Límite superior

# Valorees exactos
x1 = 41/13
x2 = -3/13
```

Se ejecuta el algoritmo PSO 30 veces para garantizar la obtención de los mejores resultados:

In [28]:
```
# Ejecutar el PSO 10 veces y almacenar los resultados
num_rondas = 30
mejores_resultados = []
```

```
for _ in range(num_rondas):
    solucion_optima, valor_optimo = pso(
        funcion_objetivo,
        lb,
        ub,
        swarmsize=num_particulas,
        maxiter=cantidad_iteraciones,
        omega=w,
        phip=c1,
        phig=c2
    )
    mejores_resultados.append((solucion_optima, valor_optimo))

# Seleccionar el mejor resultado de todas las ejecuciones
mejor_solucion, mejor_error = min(mejores_resultados, key=lambda x: x[1])

# Imprimir la mejor solución encontrada
print(f"\nMejor solución óptima (x1, x2): {mejor_solucion}")
print(f"Error mínimo encontrado: {mejor_error}")

# Diferencias relativas entre las posiciones
x1_dif = (mejor_solucion[0] - x1) * 100 / x1
x2_dif = (mejor_solucion[1] - x2) * 100 / x2

print(f"\nDiferencia relativa entre x_1 encontrado y esperado: {abs(x1_dif):.2f}%")
print(f"Diferencia relativa entre x_2 encontrado y esperado: {abs(x2_dif):.2f}%")
```

```
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08
Stopping search: Swarm best objective change less than 1e-08

Mejor solución óptima (x1, x2): [ 3.11764666 -0.17647126]
Error mínimo encontrado: 1.5170022563466804e-11

Diferencia relativa entre x_1 encontrado y esperado: 1.15%
Diferencia relativa entre x_2 encontrado y esperado: 23.53%
```

b. Transcribir en el informe los valores de x1 y x2 encontrados por el algoritmo.

Solución:

Después de aplicar el algoritmo PSO a la solución del sistema de dos ecuaciones, se oncontró que el método convergió bastante bien al mínimo global, pues el error mínimo obtenido fue muy bajo, del orden de 1e-10.

Los valores de x_1 y x_2 encontrados por el algoritmo fueron:

$$\begin{cases} x_1 \approx 3.12 \\ x_2 \approx -0.18 \end{cases}$$

Sin embargo, al comparar estos valores con los valores exactos se observa que el valor de x_1 encontrado por PSO se encuentra bastante cerca del exacto, con una diferencia relativa del 1% aproximadamente, mientras que el valor de x_2 encontrado por PSO se encuentra más alejado del exacto, con una diferencia relativa del 24% aproximadamente.

La discrepancia entre la precisión en los valores encontrados para x_1 y x_2 puede deberse a varios factores relacionados con la naturaleza del sistema de ecuaciones y el funcionamiento del algoritmo PSO.

En el caso del sistema de ecuaciones, los coeficientes de x_1 y x_2 afectan cómo el PSO explora el espacio de búsqueda. En la primera ecuación, el coeficiente de x_1 es 3 y mayor que el de x_2, lo que implica que la influencia de x_1 en la función objetivo es más significativa. En la segunda ecuación, el coeficiente de x_2 es -5 y mayor (en valor absoluto) que el de x_1, lo que hace que las variaciones en x_2 afecten más la función objetivo.

Así, el PSO puede encontrar una mejor aproximación para x_1 porque la función objetivo depende más de él en la primera ecuación. Por otro lado, las variaciones en x_2 afectan la función objetivo de una manera más complicada debido a los coeficientes conflictivos en ambas ecuaciones.

Además, la naturaleza del algoritmo PSO implica que las partículas exploran el espacio de búsqueda para minimizar el error. Si el espacio de búsqueda no está bien equilibrado o si las partículas convergen prematuramente, pueden encontrar una solución que minimiza el error global pero no necesariamente los errores individuales para cada variable.

c. Indicar la URL del repositorio en donde se encuentra el algoritmo PSO.

Solución:

Este notebook se puede encontrar en el siguiente repositorio: https://github.com/fede0ter0/ceia_algoritmos_evolutivos.git carpeta TP2.

d. Realizar observaciones/comentarios/conclusiones sobre: (i) ¿Cómo eligió los limites superior e inferior de x1 y x2?. (ii) ¿PSO puede resolver un sistema de n ecuaciones con n incógnitas no lineal?. Demostrar. (iii) ¿Cómo logró resolver el ejercicio?. (iv) ¿Los resultados obtenidos guardan relación directa con los valores de los parámetros elegidos?. Demostrar.

Solución:

Los límites superior e inferior se escogieron con base en las soluciones exactas $x_1 \approx 3.15$ y $x_2 \approx -0.23$. Con ellas se asumió que un rango entre -5 y 5 para las variables era suficiente para encontrar el mínimo global. Sin embargo, se pueden usar métodos gráficos para determinar el espacio de búsqueda.

Si se acota el espacio de búsqueda a una región adecuada y si se hacen pruebas para obtener los mejores parámetros, es posible afirmar que el algoritmo PSO puede resolver un sistema de n ecuaciones no lineales con n incógnitas, con una precisión moderada y en ocasiones alta. Esto se debe a que PSO es un algoritmo de optimización global que puede explorar grandes espacios de búsqueda y encontrar soluciones incluso para problemas no lineales complejos con múltiples mínimos locales.

Además, a diferencia de los métodos basados en gradientes, PSO no necesita información de derivadas de las funciones, lo que lo hace adecuado para sistemas de ecuaciones no lineales y no diferenciables. Por ende, PSO puede ser aplicado a cualquier problema donde se pueda definir una función objetivo. En el caso de sistemas de ecuaciones, esto es posible al expresar el sistema como una función que mide el error global.

No obstante, PSO encuentra soluciones aproximadas que minimizan el error, pero no garantiza encontrar una solución exacta. El éxito de PSO depende de una buena elección de parámetros principalmente del número de partículas, la cantidad de iteraciones y los coeficientes cognitivos y sociales.

Por este motivo, los resultados obtenidos guardan relación directa con los parámetros escogidos. Por ejemplo, más partículas e iteraciones, ayudan al método a explorar más puntos del espacio de búsqueda y encontrar más fácil el mínimo global.

Coeficientes cognitivos y sociales muy altos o muy bajos aumentan la incertidumbre en el valor de las soluciones a encontrar, como se muestra a continuación:

In [29]:
```
# Ejecutar el PSO 10 veces y almacenar los resultados
num_rondas = 30
mejores_resultados = []
```

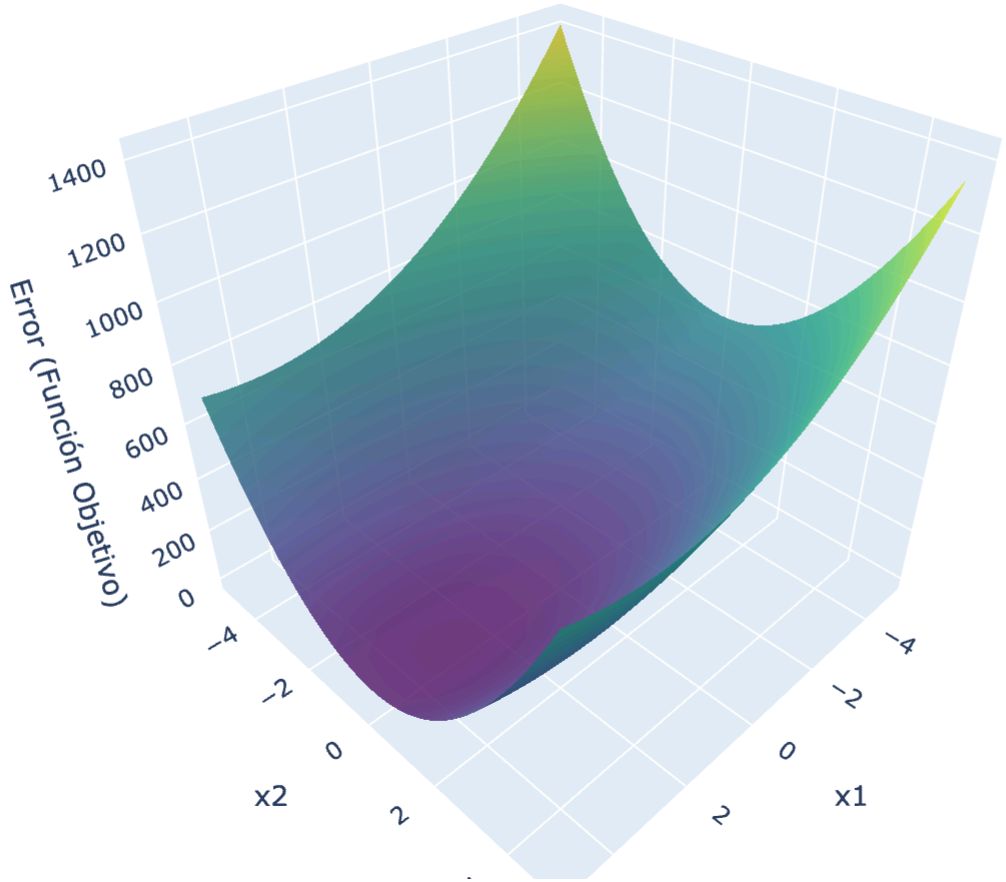
```
for _ in range(num_rondas):
    solucion_optima, valor_optimo = pso(
        funcion_objetivo,
        lb,
        ub,
        swarmsize=num_particulas,
        maxiter=cantidad_iteraciones,
        omega=w,
        phip=0,
        phig=0
    )
    mejores_resultados.append((solucion_optima, valor_optimo))

# Seleccionar el mejor resultado de todas las ejecuciones
mejor_solucion, mejor_error = min(mejores_resultados, key=lambda x: x[1])

# Imprimir la mejor solución encontrada
print(f"\nMejor solución óptima (x1, x2): {mejor_solucion}")
print(f"Error mínimo encontrado: {mejor_error}")

# Diferencias relativas entre las posiciones
x1_dif = (mejor_solucion[0] - x1) * 100 / x1
x2_dif = (mejor_solucion[1] - x2) * 100 / x2

print(f"\nDiferencia relativa entre x_1 encontrado y esperado: {abs(x1_dif):.2f}%")
print(f"Diferencia relativa entre x_2 encontrado y esperado: {abs(x2_dif):.2f}%")
```

```
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50
Stopping search: maximum iterations reached --> 50

Mejor solución óptima (x1, x2): [ 3.04724447 -0.08169751]
Error mínimo encontrado: 0.2966968707028284

Diferencia relativa entre x_1 encontrado y esperado: 3.38%
Diferencia relativa entre x_2 encontrado y esperado: 64.60%
```

Sin embargo, debido a la presencia de un mínimo global característico en la región considerada, como se muestra en el gráfico 3D a continuación, el parámetro w no tiene un efecto importante en el rendimiento del algoritmo, por lo que es posible anularlo para este caso:

In [30]:
```
# Definir el rango para graficar la función
x1_vals = np.linspace(lb[0], ub[0], 400)
x2_vals = np.linspace(lb[0], ub[0], 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
Z = funcion_objetivo([X1, X2])

# Crear el gráfico 3D interactivo
fig = go.Figure(data=[go.Surface(z=Z, x=X1, y=X2, colorscale='viridis', opacity=0.8)])

# Ajustar etiquetas y título
fig.update_layout(
    title='Función Objetivo 3D Interactiva',
    scene=dict(
        xaxis_title='x1',
        yaxis_title='x2',
        zaxis_title='Error (Función Objetivo)'
    ),
    margin=dict(l=0, r=0, b=0, t=40)
)

# Mostrar el gráfico
fig.show()
```

Función Objetivo 3D Interactiva



In [ ]: