Práctica Calificada N° 2

Considere el siguiente problema de optimización:

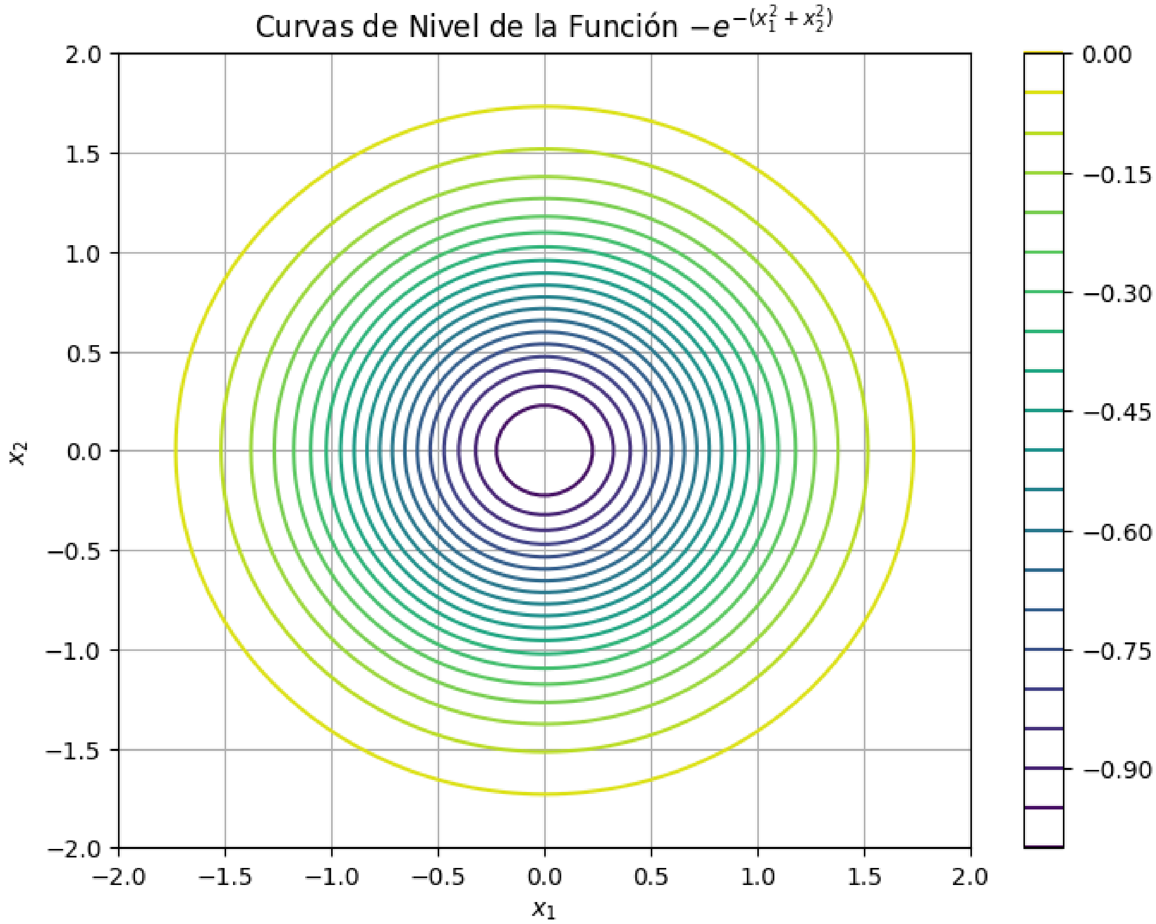
$$\min\left\{-e^{-x_1^2-x_2^2} : (x_1 \cdot x_2) \in \mathbb{R}^2\right\}$$

1. (2 puntos) Grafique las curvas de nivel de la función

$$f(x_1, x_2) = -e^{-x_1^2-x_2^2}$$

In [ ]:
```
pip install sympy matplotlib numpy pandas
```

In [ ]:
```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Definimos las variables simbólicas
x1, x2 = sp.symbols('x1 x2')

# Definimos la función a optimizar
opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))

# Convertimos la función sympy a una función numpy para evaluación eficiente
opti_np = sp.lambdify((x1, x2), opti, 'numpy')

# Creamos una cuadrícula de valores para x1 y x2
x1_vals = np.linspace(-2, 2, 100)
x2_vals = np.linspace(-2, 2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluamos la función en cada punto de la cuadrícula
Z = opti_np(X1, X2)

# Creamos la gráfica de las curvas de nivel
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, Z, levels=20, cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de la Función $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.grid(True)
plt.show()
```



4. (2 puntos) Aplicar el método del gradiente con búsqueda exacta y de Armijo.

Algoritmo busqueda Gradiente - Exacta

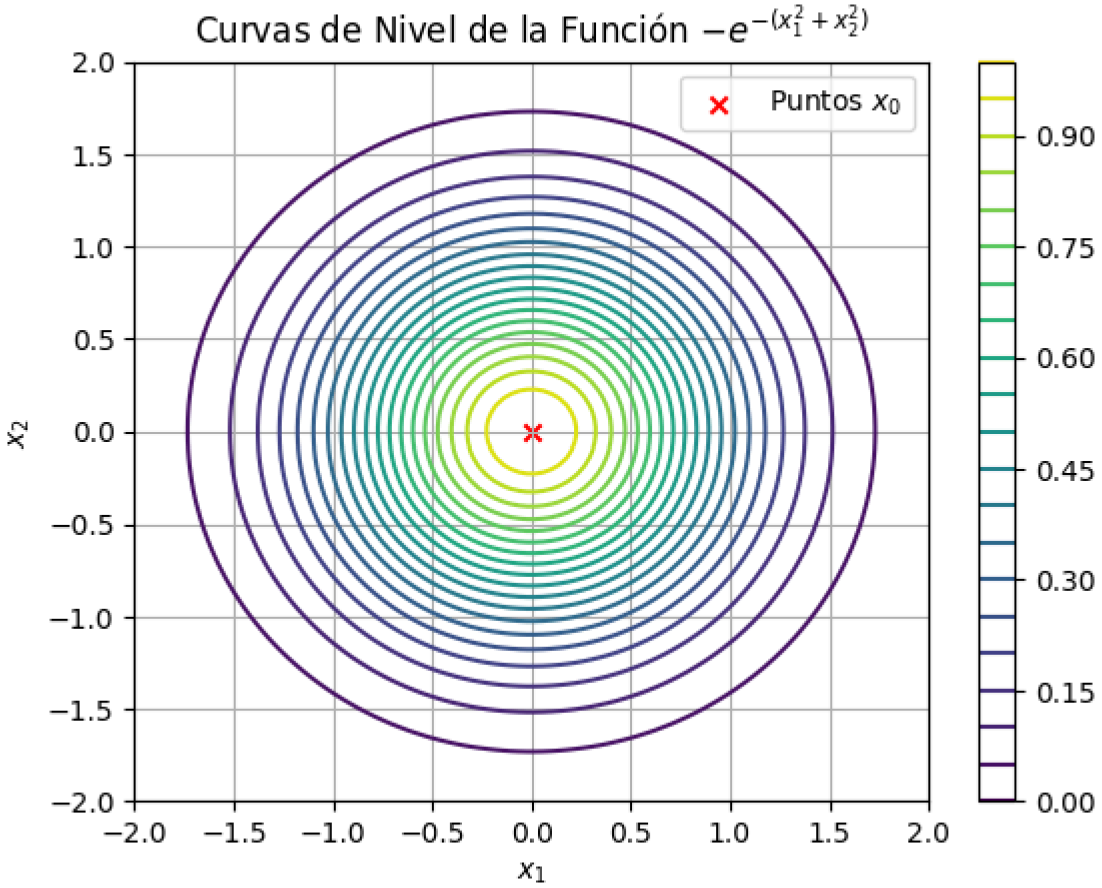
In [ ]:
```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# Definimos las variables simbólicas
x1, x2 = sp.symbols('x1 x2')

# Definimos la función a optimizar
opti = -sp.exp(-(x1 ** 2) - (x2 ** 2))

# Variables iniciales
x0 = np.array([-1.2, 1], dtype=np.float64)
k = 1
error = 1e-5

# Lista que almacenará los datos de cada iteración
data = []

def busqueda_exacta(x0, opti, grad):
    alpha = sp.symbols('alpha')
    x_new = x0 - alpha * grad
    func_new = opti.subs({x1: x_new[0], x2: x_new[1]})
    dfunc_new = sp.diff(func_new, alpha)
```

```python
    # Encontrar la solución de alpha que minimiza la función
    alpha_opt = sp.solve(dfunc_new, alpha)
    alpha_opt = [sol.evalf() for sol in alpha_opt if sol.is_real and sol > 0]

    if alpha_opt:
        return min(alpha_opt)
    else:
        return 1  # Valor de fallback en caso de que no se encuentre un valor óp

# Iniciar el contador de tiempo
start_time = time.time()

while k < 101:
    # Obtener el gradiente de la función
    grad = np.array([float(opti.diff(var).subs({x1: x0[0], x2: x0[1]})) for var

    # Obtener la norma del gradiente
    Normadf = np.linalg.norm(grad)

    # Condición de terminación del algoritmo
    if Normadf < error:
        break

    # Obtener el valor óptimo de alpha mediante búsqueda exacta
    alpha = busqueda_exacta(x0, opti, grad)

    # Actualizar el valor de x0
    x0 = x0 - float(alpha) * grad

    # Calcular el valor de la función en x0
    fun = float(opti.subs({x1: x0[0], x2: x0[1]}))

    # Calcular el error
    mod = np.linalg.norm(x0 - np.array([0, 0]))

    # Añadir los datos de la iteración actual a la lista
    data.append([k, float(alpha), x0.tolist(), fun, mod])

    k += 1

# Crear un DataFrame de pandas con los datos recolectados
df = pd.DataFrame(data, columns=['Iteración', 'Alpha', 'x0', 'f(x0)', '||x0 - x|

# Imprimir la tabla
display(df)

# Imprimir el tiempo de ejecución del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-2, 2, 100)
x2_vals = np.linspace(-2, 2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la función en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el gráfico de las curvas de nivel
```

```
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Función $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Pu

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```

	Iteración	Alpha	x0	f(x0)	||x0 - x||
0	1	5.73652	[4.440892098500626e-16, -2.220446049250313e-16]	-1.0	4.965068e-16

```
Tiempo de ejecución: 0.249054 segundos
```



Algoritmo busqueda Gradiente - Armijo

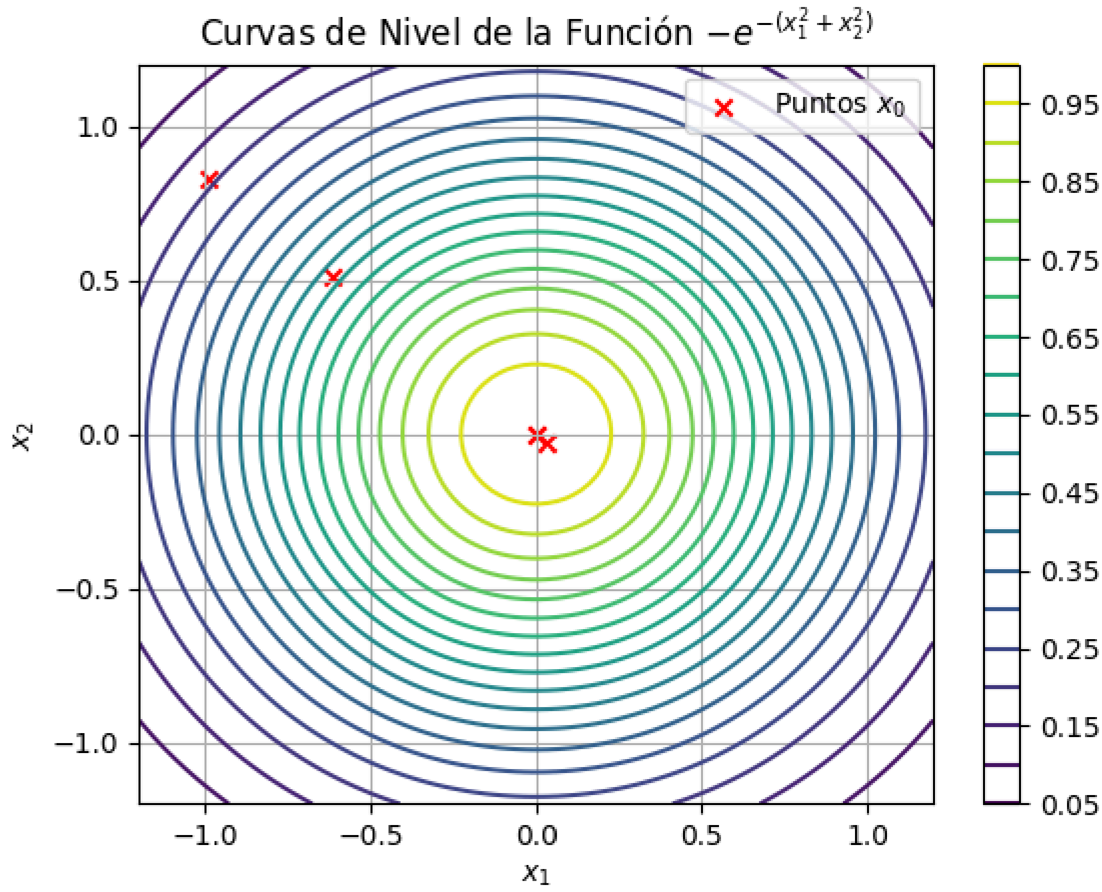
In [ ]:
```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
```

```
# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([-1.2, 1], dtype=np.float64)
k = 1
error = 1e-5

data = []

def armijo(x0, opti):
  b, s, o, k1, m = 0.5, 1, 0.1, 1, 0

  while True:
    # Hallamos el valor de lambda
    lmb = s * (b ** m)
    # Hallamos f(x0)
    f = float(opti.subs(zip((x1, x2), x0)))
    # Hallamos el valor de la gradiente
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x
    # Hallamos el valor de f(x0 - lmb * grad)
    f_k = float(opti.subs(zip((x1, x2), x0 - lmb * grad)))
    mod = grad[0] ** 2 + grad[1] ** 2

    # Condición de parada
    if f_k <= f - o * lmb * mod:
      break
    else:
      m += 1
      k1 += 1

  return k1, lmb

# Iniciar el contador de tiempo
start_time = time.time()

while k < 101:
  # Obtener la gradiente de la función
  grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x1,

  # Obtener la norma del gradiente
  Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

  # Condición para finalizar el programa
  if Normadf < error:
    break

  # Obtener el número de iteraciones internas y el valor de lambda
  iter, arm = armijo(x0, opti)

  # Actualizar el valor de x0
  x0 = x0 - arm * grad

  # Hallar el valor de la función en x0
  fun = float(opti.subs(zip((x1, x2), x0)))

  # Error
  mod = np.linalg.norm(x0 - (0,0))
```

```python
    # Introducir el número de iteraciones, iteraciones internas, lambda, x0 y fun(
    data.append([k, iter, x0, fun, mod])

    k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data, columns=['Iteración', 'Iteraciones Internas', 'x0', 'f(x

# Imprimir la información obtenida
display(df)

# Imprimir el tiempo de ejecución del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-1.2, 1.2, 100)
x2_vals = np.linspace(-1.2, 1.2, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la función en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el gráfico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Función $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Pu

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```

	Iteración	Iteraciones Internas	x0	f(x0)	|| x0 - x||
0	1	1	[-0.9908139564912448, 0.8256782970760375]	-0.189483	1.289751e+00
1	2	1	[-0.6153295220274321, 0.5127746016895267]	-0.526466	8.009795e-01
2	3	1	[0.0325708257491395, -0.027142354790949508]	-0.998204	4.239771e-02
3	4	2	[5.849562203281283e-05, -4.874635169400954e-05]	-1.000000	7.614424e-05
4	5	2	[3.3915437054913775e-13, -2.826286432536587e-13]	-1.000000	4.414801e-13

```
Tiempo de ejecución: 0.055907 segundos
```



5. (2 puntos) Aplicar el método de Newton puro y con búsqueda exacta y de Armijo

Algoritmo Newton Puro - Armijo

In [ ]:

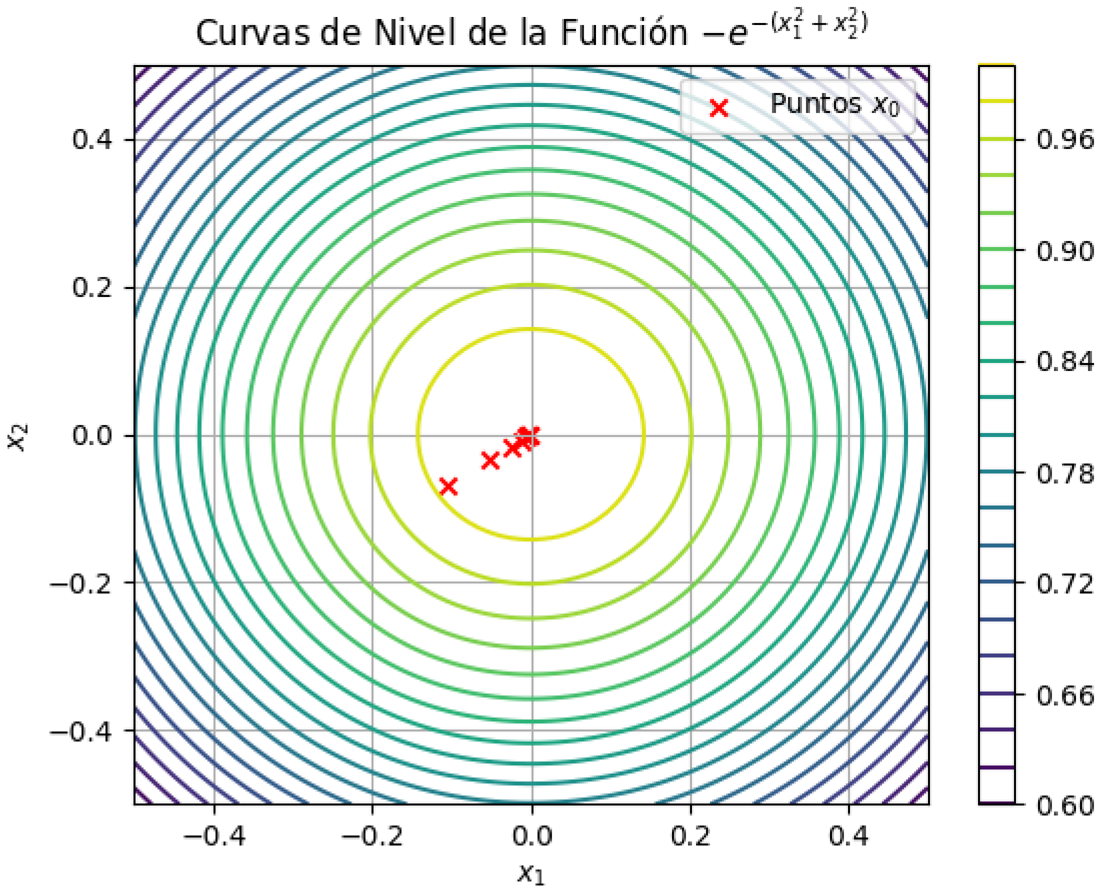
```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([0.3, 0.2], dtype=np.float64)
k = 1
error = 1e-3
data1 = []

# Crearemos la función armijo
def armijo(x0, opti):
    b, s, o, k1, m = 0.5, 1, 0.1, 1, 0

    while True:
        # Hallamos el valor de lambda
        lmb = s * (b ** m)
        # Hallamos f(x0)
        f = float(opti.subs(zip((x1, x2), x0)))
        # Hallamos el valor de la gradiente
```

```
        grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var i
        # Hallamos el valor de f(x0 - lmb * grad)
        f_k = float(opti.subs(zip((x1, x2), x0 - lmb * grad)))
        mod = grad[0] ** 2 + grad[1] ** 2

        # Condición de parada
        if f_k <= f - o * lmb * mod:
            break
        else:
            m += 1
            k1 += 1

    return k1, lmb

# Iniciar el contador de tiempo
start_time = time.time()

# Método de Newton
while k < 101:
    # Obtenemos la gradiente de la función
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x

    # Calcular la hessiana simbólicamente
    hess = sp.hessian(opti, (x1, x2))

    # Sustituir los valores de x0 en la hessiana
    hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)

    # Obtenemos la norma de la función
    Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

    if Normadf < error:
        break

    # Hallamos el valor de d, mediante la solución del sistema lineal
    d = np.linalg.solve(hess_x0, -grad)

    # Obtendremos el número de iteraciones internas y el valor de lambda
    iter, arm = armijo(x0, opti)

    # Actualizaremos x0
    x0 = x0 + arm * d

    # Hallaremos el valor de la función en x0
    fun = float(opti.subs(zip((x1, x2), x0)))

    # Error
    mod = np.linalg.norm(x0 - (0,0))

    # Introduciremos el número de iteraciones, iteraciones internas, lambda, x0
    data1.append([k, iter, x0, fun, mod])

    k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data1, columns=['k', 'Iteración interna', 'x0', 'fun(x0)', '||

# Imprimir la información obtenida
display(df)
```

```python
# Imprimir el tiempo de ejecución del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-0.5, 0.5, 100)
x2_vals = np.linspace(-0.5, 0.5, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la función en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))

# Crear el gráfico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Función $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Pu

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```

	k	Iteración interna	x0	fun(x0)	|| x0 - x||
0	1	1	[-0.10540540540540538, -0.07027027027027027]	-0.984080	0.126682
1	2	2	[-0.05095504089360971, -0.03397002726240648]	-0.996257	0.061240
2	3	2	[-0.025284975501293144, -0.016856650334195433]	-0.999077	0.030389
3	4	2	[-0.012619094459946219, -0.00841272963996415]	-0.999770	0.015166
4	5	2	[-0.0063066433059358615, -0.0042044288706239085]	-0.999943	0.007580
5	6	2	[-0.0031529592885440974, -0.002101972859029399]	-0.999986	0.003789
6	7	2	[-0.0015764343682351865, -0.0010509562454901248]	-0.999996	0.001895
7	8	2	[-0.0007882115252224833, -0.0005254743501483224]	-0.999999	0.000947
8	9	2	[-0.00039410505526839675, -0.00026273670351226...	-1.000000	0.000474

```
Tiempo de ejecución: 0.182519 segundos
```



Algoritmo Newton Puro - Busqueda Exacta

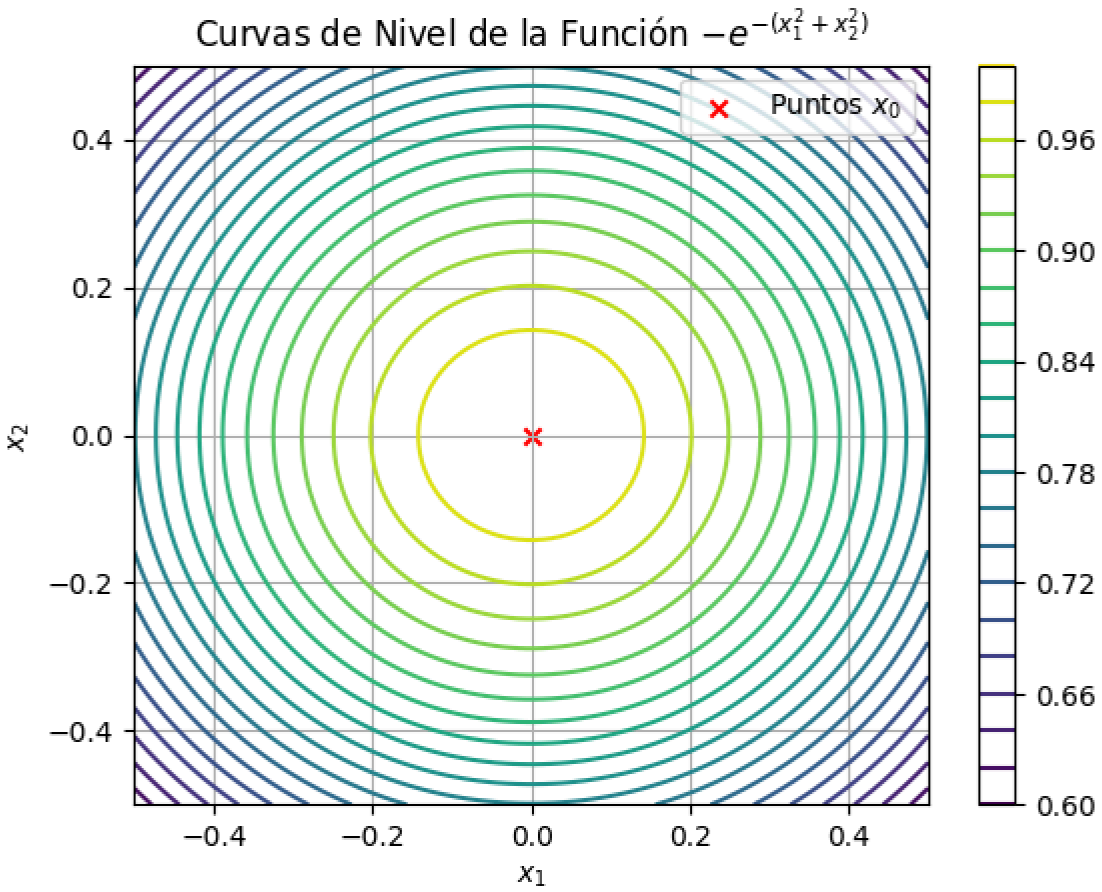
In [ ]:
```
import sympy as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from scipy.optimize import minimize_scalar

# Definimos el polinomio a optimizar
x1, x2 = sp.symbols('x1 x2')
opti = -sp.exp(-((x1 ** 2) + (x2 ** 2)))

# Variables iniciales
x0 = np.array([1, 2], dtype=np.float64)
k = 1
error = 1e-3
data1 = []

# Método de búsqueda unidimensional para encontrar el mejor lambda
def busqueda_unidimensional(x0, d, opti):
    # Definimos la función en términos de lambda
    lambda_var = sp.symbols('lambda')
    x_new = x0 + lambda_var * d
    f_lambda = opti.subs(zip((x1, x2), x_new))

    # Convertimos f_lambda a una función de numpy
    f_lambda_func = sp.lambdify(lambda_var, f_lambda, 'numpy')

    # Usamos minimize_scalar para encontrar el valor óptimo de lambda
    res = minimize_scalar(f_lambda_func)
```

```python
    return res.x

# Iniciar el contador de tiempo
start_time = time.time()

# Método de Newton
while k < 101:
    # Obtenemos la gradiente de la función
    grad = np.array([float(opti.diff(var).subs(zip((x1, x2), x0))) for var in (x

    # Calcular la hessiana simbólicamente
    hess = sp.hessian(opti, (x1, x2))

    # Sustituir los valores de x0 en la hessiana
    hess_x0 = np.array(hess.subs(zip((x1, x2), x0))).astype(np.float64)

    # Obtenemos la norma de la función
    Normadf = (grad[0]**2 + grad[1]**2)**(0.5)

    if Normadf < error:
        break

    # Hallamos el valor de d, mediante la solución del sistema lineal
    d = np.linalg.solve(hess_x0, -grad)

    # Obtendremos el valor de lambda usando la búsqueda unidimensional
    arm = busqueda_unidimensional(x0, d, opti)

    # Actualizaremos x0
    x0 = x0 + arm * d

    # Hallaremos el valor de la función en x0
    fun = float(opti.subs(zip((x1, x2), x0)))

    # Error
    mod = np.linalg.norm(x0 - (0, 0))

    # Introduciremos el número de iteraciones, lambda, x0 y fun(x0)
    data1.append([k, 1, x0, fun, mod])

    k += 1

# Crear DataFrame de pandas
df = pd.DataFrame(data1, columns=['k', 'Iteración interna', 'x0', 'fun(x0)', '||

# Imprimir la información obtenida
display(df)

# Imprimir el tiempo de ejecución del programa
execution_time = time.time() - start_time
print(f"Tiempo de ejecución: {execution_time:.6f} segundos")

# Crear una malla de valores para x1 y x2
x1_vals = np.linspace(-0.5, 0.5, 100)
x2_vals = np.linspace(-0.5, 0.5, 100)
X1, X2 = np.meshgrid(x1_vals, x2_vals)

# Evaluar la función en cada punto de la malla
Z = np.exp(-(X1 ** 2 + X2 ** 2))
```

```python
# Crear el gráfico de las curvas de nivel
plt.contour(X1, X2, Z, levels=20)
plt.colorbar()
plt.grid()
plt.title('Curvas de Nivel de la Función $-e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Extraer y graficar los puntos x0 del DataFrame
x0_points = np.array([point for point in df['x0']])
plt.scatter(x0_points[:, 0], x0_points[:, 1], color='red', marker='x', label='Pu

# Añadir leyenda
plt.legend()

# Mostrar la gráfica
plt.show()
```

	k	Iteración interna	x0	fun(x0)	|| x0 - x||
0	1	1	[-3.858942942969179e-09, -7.717884553670729e-09]	-1.0	8.628858e-09

```
Tiempo de ejecución: 0.022834 segundos
```



6. (2 puntos) Aplicar el método del Cuasi-Newton con búsqueda exacta y de Armijo.

Algoritmo Cuasi-Newton - Comparativa : Busqueda Exacta - Armijo

In [ ]:

```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import sympy as sp

# Definir la función y su gradiente
def f(x):
    return -np.exp(-(x[0]**2 + x[1]**2))

def grad_f(x):
    exp_term = np.exp(-(x[0]**2 + x[1]**2))
    return np.array([2 * x[0] * exp_term, 2 * x[1] * exp_term])

# Búsqueda exacta usando scipy.optimize
def exact_line_search(x, d):
    phi = lambda alpha: f(x + alpha * d)
    result = minimize_scalar(phi)
    return result.x

# Método del Cuasi-Newton con Búsqueda Exacta (BFGS)
def quasi_newton_bfgs_exact(x0, tol=1e-6, max_iter=1000):
    x = x0
    n = len(x)
    H = np.eye(n)  # Inicializar H como la identidad
    iter_count = 0
    data = []

    for _ in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break

        d = -H @ grad
        alpha = exact_line_search(x, d)
        x_new = x + alpha * d

        s = x_new - x
        y = grad_f(x_new) - grad

        if np.dot(y, s) > 0:  # Para asegurar que la actualización sea positiva
            rho = 1.0 / np.dot(y, s)
            I = np.eye(n)
            H = (I - rho * np.outer(s, y)) @ H @ (I - rho * np.outer(y, s)) + rh

        x = x_new
        data.append([iter_count, 1, x, f(x), np.linalg.norm(x)])
        iter_count += 1

    return x, iter_count, data

# Método del Cuasi-Newton con el Criterio de Armijo (BFGS)
def quasi_newton_bfgs_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, sig
    x = x0
    n = len(x)
    H = np.eye(n)  # Inicializar H como la identidad
    iter_count = 0
    total_internal_iter_count = 0
    data = []
```

```
    for _ in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break

        d = -H @ grad
        t = alpha
        internal_iter_count = 0

        while f(x + t * d) > f(x) + sigma * t * np.dot(grad, d):
            t *= beta
            internal_iter_count += 1

        x_new = x + t * d
        s = x_new - x
        y = grad_f(x_new) - grad

        if np.dot(y, s) > 0:  # Para asegurar que la actualización sea positiva
            rho = 1.0 / np.dot(y, s)
            I = np.eye(n)
            H = (I - rho * np.outer(s, y)) @ H @ (I - rho * np.outer(y, s)) + rh

        x = x_new
        data.append([iter_count, internal_iter_count, x, f(x), np.linalg.norm(x)
        iter_count += 1
        total_internal_iter_count += internal_iter_count

    return x, iter_count, total_internal_iter_count, data

# Inicialización y llamada al método con búsqueda exacta
x0 = np.array([1.0, 1.0])
result_exact, iter_count_exact, data_exact = quasi_newton_bfgs_exact(x0)
print("Resultado (Cuasi-Newton con Búsqueda Exacta):", result_exact)
print("Número de iteraciones:", iter_count_exact)

# Inicialización y llamada al método con Armijo
result_armijo, iter_count_armijo, total_internal_iter_count_armijo, data_armijo
print("Resultado (Cuasi-Newton con Armijo):", result_armijo)
print("Número de iteraciones externas:", iter_count_armijo)
print("Número de iteraciones internas:", total_internal_iter_count_armijo)

# Crear un DataFrame de pandas para ambos métodos
import pandas as pd

df_exact = pd.DataFrame(data_exact, columns=['Iteración', 'Iteraciones internas'
df_armijo = pd.DataFrame(data_armijo, columns=['Iteración', 'Iteraciones interna

# Imprimir los DataFrames
print("Método de Búsqueda Exacta:")
display(df_exact)

print("Método de Armijo:")
display(df_armijo)

# Graficar las curvas de nivel
x1_vals = np.linspace(-2, 2, 400)
x2_vals = np.linspace(-2, 2, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
F = -np.exp(-(X1**2 + X2**2))
```

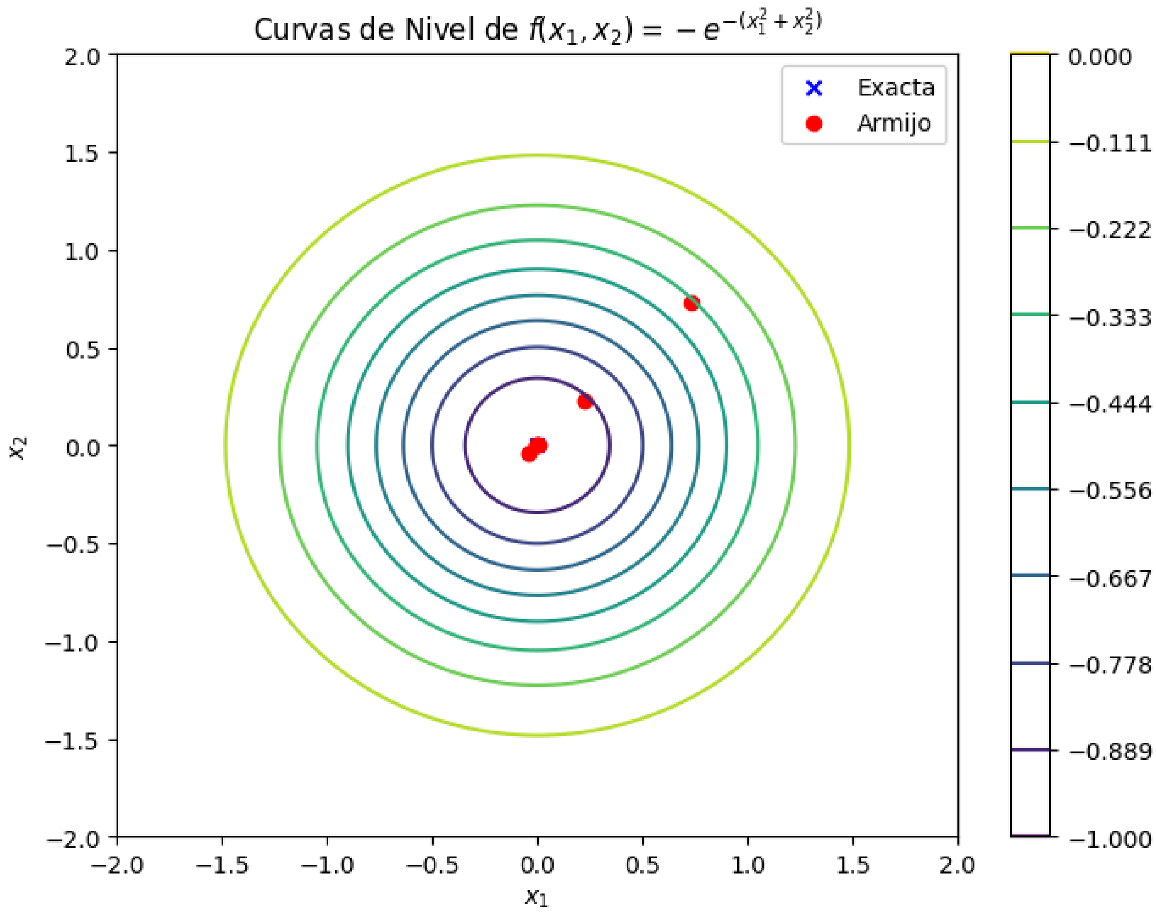
```python
plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de $f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Graficar los puntos para ambos métodos
x0_points_exact = np.array([point[2] for point in data_exact])
x0_points_armijo = np.array([point[2] for point in data_armijo])
plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='
plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker=

plt.legend()
plt.show()
```

```
Resultado (Cuasi-Newton con Búsqueda Exacta): [7.93888733e-10 7.93888733e-10]
Número de iteraciones: 1
Resultado (Cuasi-Newton con Armijo): [3.36773197e-10 3.36773190e-10]
Número de iteraciones externas: 6
Número de iteraciones internas: 3
Método de Búsqueda Exacta:
```

	Iteración	Iteraciones internas	x0	f(x0)	||x0 - x||
0	0	1	[7.938887325309452e-10, 7.938887325309452e-10]	-1.0	1.122728e-09

```
Método de Armijo:
```

	Iteración	Iteraciones internas	x0	f(x0)	||x0 - x||
0	0	0	[0.7293294335267746, 0.7293294335267746]	-0.345127	1.031428e+00
1	1	0	[0.22590689055997926, 0.22590689055997926]	-0.902968	3.194806e-01
2	2	3	[-0.043062671899500304, -0.043062671899500304]	-0.996298	6.089981e-02
3	3	0	[0.0036774538987604105, 0.0036774538987604036]	-0.999973	5.200705e-03
4	4	0	[-1.249353007418217e-05, -1.2493530074175231e-05]	-1.000000	1.766852e-05
5	5	0	[3.367731971182339e-10, 3.3677319023185605e-10]	-1.000000	4.762692e-10



7. (2 puntos) Aplicar el método del gradiente conjugado con búsqueda unidimensional y de Armijo

In [ ]:
```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import sympy as sp

# Definir la función y su gradiente
def f(x):
    return -np.exp(-(x[0]**2 + x[1]**2))

def grad_f(x):
    exp_term = np.exp(-(x[0]**2 + x[1]**2))
    return np.array([2 * x[0] * exp_term, 2 * x[1] * exp_term])

# Búsqueda exacta usando scipy.optimize
def exact_line_search(x, d):
    phi = lambda alpha: f(x + alpha * d)
    result = minimize_scalar(phi)
    return result.x

# Método del Gradiente Conjugado con Búsqueda Exacta
def conjugate_gradient_exact(x0, tol=1e-6, max_iter=1000):
    x = x0
    grad = grad_f(x)
    d = -grad
    iter_count = 0
    data = []
```

```
    for _ in range(max_iter):
        if np.linalg.norm(grad) < tol:
            break
        alpha = exact_line_search(x, d)
        x_new = x + alpha * d
        grad_new = grad_f(x_new)
        beta = np.dot(grad_new, grad_new) / np.dot(grad, grad)
        d = -grad_new + beta * d
        x = x_new
        grad = grad_new
        iter_count += 1
        data.append([iter_count, 1, x, f(x), np.linalg.norm(x)])

    return x, iter_count, data

# Método del Gradiente Conjugado con el Criterio de Armijo
def conjugate_gradient_armijo(x0, tol=1e-6, max_iter=1000, alpha=1, beta=0.5, si
    x = x0
    grad = grad_f(x)
    d = -grad
    iter_count = 0
    total_internal_iter_count = 0
    data = []

    for _ in range(max_iter):
        if np.linalg.norm(grad) < tol:
            break
        t = alpha
        internal_iter_count = 0
        while f(x + t * d) > f(x) + sigma * t * np.dot(grad, d):
            t *= beta
            internal_iter_count += 1

        x_new = x + t * d
        grad_new = grad_f(x_new)
        beta_cg = np.dot(grad_new, grad_new) / np.dot(grad, grad)
        d = -grad_new + beta_cg * d
        x = x_new
        grad = grad_new
        iter_count += 1
        total_internal_iter_count += internal_iter_count
        data.append([iter_count, internal_iter_count, x, f(x), np.linalg.norm(x)

    return x, iter_count, total_internal_iter_count, data

# Inicialización y llamada al método con búsqueda exacta
x0 = np.array([1.0, 1.0])
result_exact, iter_count_exact, data_exact = conjugate_gradient_exact(x0)
print("Resultado (Gradiente Conjugado con Búsqueda Exacta):", result_exact)
print("Número de iteraciones:", iter_count_exact)

# Inicialización y llamada al método con Armijo
result_armijo, iter_count_armijo, total_internal_iter_count_armijo, data_armijo
print("Resultado (Gradiente Conjugado con Armijo):", result_armijo)
print("Número de iteraciones externas:", iter_count_armijo)
print("Número de iteraciones internas:", total_internal_iter_count_armijo)

# Crear un DataFrame de pandas para ambos métodos
import pandas as pd
```

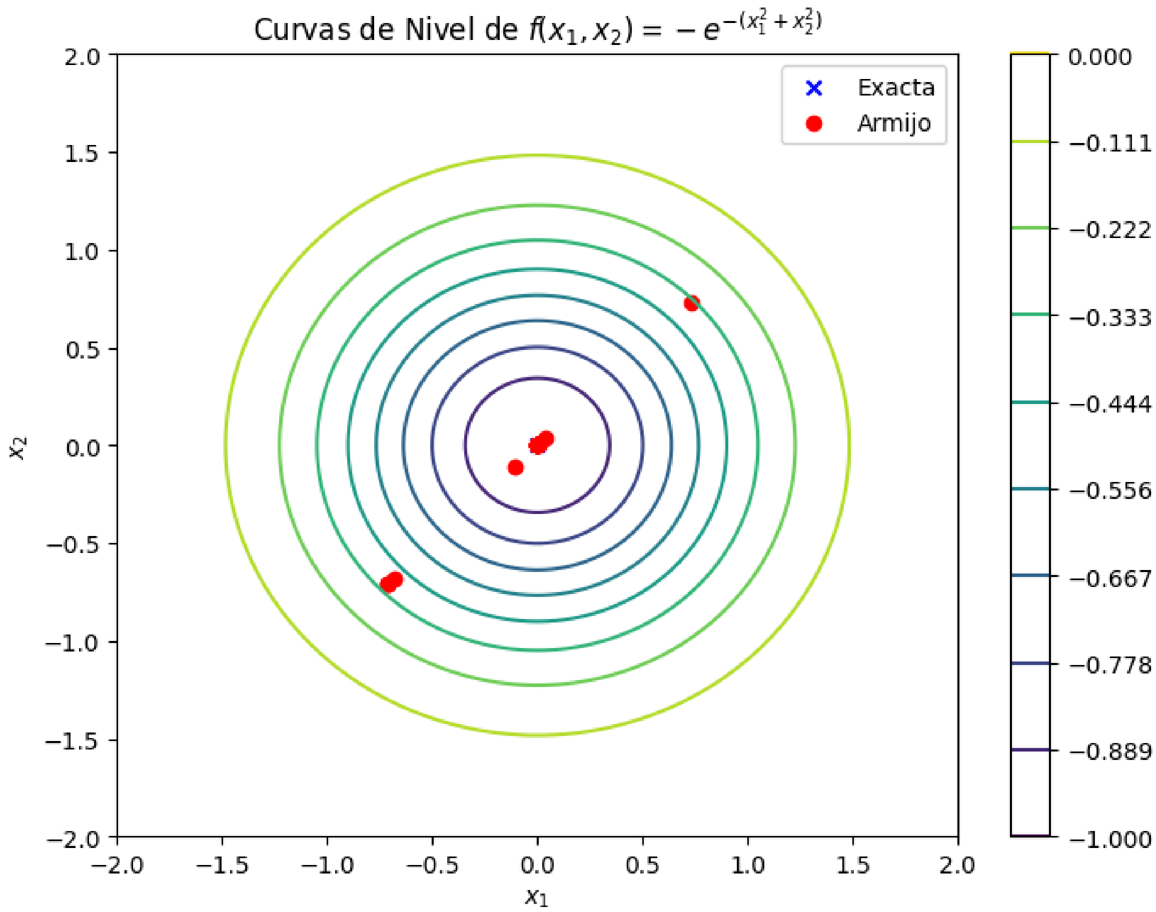
```
df_exact = pd.DataFrame(data_exact, columns=['Iteración', 'Iteraciones internas'
df_armijo = pd.DataFrame(data_armijo, columns=['Iteración', 'Iteraciones interna

# Imprimir los DataFrames
print("Método de Búsqueda Exacta:")
display(df_exact)

print("Método de Armijo:")
display(df_armijo)

# Graficar las curvas de nivel
x1_vals = np.linspace(-2, 2, 400)
x2_vals = np.linspace(-2, 2, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
F = -np.exp(-(X1**2 + X2**2))

plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de $f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Graficar los puntos para ambos métodos
x0_points_exact = np.array([point[2] for point in data_exact])
x0_points_armijo = np.array([point[2] for point in data_armijo])
plt.scatter(x0_points_exact[:, 0], x0_points_exact[:, 1], color='blue', marker='
plt.scatter(x0_points_armijo[:, 0], x0_points_armijo[:, 1], color='red', marker=

plt.legend()
plt.show()
```

```
Resultado (Gradiente Conjugado con Búsqueda Exacta): [7.93888733e-10 7.93888733e-
10]
Número de iteraciones: 1
Resultado (Gradiente Conjugado con Armijo): [3.48047568e-07 3.48047568e-07]
Número de iteraciones externas: 16
Número de iteraciones internas: 112
Método de Búsqueda Exacta:
```

	Iteración	Iteraciones internas	x0	f(x0)	||x0 - x||
0	1	1	[7.938887325309452e-10, 7.938887325309452e-10]	-1.0	1.122728e-09

```
Método de Armijo:
```

	Iteración	Iteraciones internas	x0	f(x0)	||x0 - x||
0	1	0	[0.7293294335267746, 0.7293294335267746]	-0.345127	1.031428e+00
1	2	0	[-0.7104130797616721, -0.7104130797616721]	-0.364447	1.004676e+00
2	3	55	[-0.7104130797616721, -0.7104130797616721]	-0.364447	1.004676e+00
3	4	53	[-0.7104130797616721, -0.7104130797616721]	-0.364447	1.004676e+00
4	5	0	[-0.6802129584463044, -0.6802129584463044]	-0.396381	9.619664e-01
5	6	0	[-0.10821429968370222, -0.10821429968370222]	-0.976851	1.530381e-01
6	7	1	[0.04145683785747126, 0.04145683785747126]	-0.996569	5.862882e-02
7	8	0	[0.004552115239788761, 0.004552115239788761]	-0.999959	6.437663e-03
8	9	1	[-0.0002238050139070965, -0.0002238050139070965]	-1.000000	3.165081e-04
9	10	0	[0.0002007143221634751, 0.0002007143221634751]	-1.000000	2.838529e-04
10	11	0	[0.0001407258660028434, 0.0001407258660028434]	-1.000000	1.990164e-04
11	12	1	[-1.4744453991543408e-05, -1.4744453991543408e...	-1.000000	2.085181e-05
12	13	0	[1.1331055020766559e-05, 1.1331055020766559e-05]	-1.000000	1.602453e-05
13	14	0	[4.06878122270987e-06, 4.06878122270987e-06]	-1.000000	5.754126e-06
14	15	1	[-4.68199130923443e-07, -4.68199130923443e-07]	-1.000000	6.621336e-07
15	16	0	[3.4804756822835973e-07, 3.4804756822835973e-07]	-1.000000	4.922136e-07



8. (2 puntos) Aplicar el método del punto proximal donde λ_k es cualquier sucesión positiva y acotada que puede ser tomado como $1/k$

In [ ]:
```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import sympy as sp

# Definir la función y su gradiente
def f(x):
    return -np.exp(-(x[0]**2 + x[1]**2))

def grad_f(x):
    exp_term = np.exp(-(x[0]**2 + x[1]**2))
    return np.array([2 * x[0] * exp_term, 2 * x[1] * exp_term])

# Búsqueda exacta usando scipy.optimize
def exact_line_search(x, d):
    phi = lambda alpha: f(x + alpha * d)
    result = minimize_scalar(phi)
    return result.x

# Método del Punto Proximal
def proximal_point_method(f, grad_f, x0, tol=1e-6, max_iter=1000, lambda_seq=Non
    x = x0
    iter_count = 0
    data = []

    if lambda_seq is None:
        lambda_seq = [1/(k+1) for k in range(max_iter)]
```

```
    for k in range(max_iter):
        grad = grad_f(x)
        if np.linalg.norm(grad) < tol:
            break

        lambda_k = lambda_seq[k]
        d = -grad
        alpha = exact_line_search(x, d)
        x_new = x + alpha * lambda_k * d

        x = x_new
        iter_count += 1
        data.append([iter_count, 1, x, f(x), np.linalg.norm(x)])

    return x, iter_count, data

# Inicialización y llamada al método del Punto Proximal
x0 = np.array([1.0, 1.0])
result_proximal, iter_count_proximal, data_proximal = proximal_point_method(f, g
print("Resultado (Método del Punto Proximal):", result_proximal)
print("Número de iteraciones:", iter_count_proximal)

# Crear un DataFrame de pandas para el método del Punto Proximal
import pandas as pd

df_proximal = pd.DataFrame(data_proximal, columns=['Iteración', 'Iteraciones int

# Imprimir el DataFrame
print("Método del Punto Proximal:")
display(df_proximal)

# Graficar las curvas de nivel
x1_vals = np.linspace(-2, 2, 400)
x2_vals = np.linspace(-2, 2, 400)
X1, X2 = np.meshgrid(x1_vals, x2_vals)
F = -np.exp(-(X1**2 + X2**2))

plt.figure(figsize=(8, 6))
contour = plt.contour(X1, X2, F, levels=np.linspace(-1, 0, 10), cmap='viridis')
plt.colorbar(contour)
plt.title('Curvas de Nivel de $f(x_1, x_2) = -e^{-(x_1^2 + x_2^2)}$')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Graficar los puntos para el método del Punto Proximal
x0_points_proximal = np.array([point[2] for point in data_proximal])
plt.scatter(x0_points_proximal[:, 0], x0_points_proximal[:, 1], color='red', mar

plt.legend()
plt.show()
```

```
Resultado (Método del Punto Proximal): [7.93888733e-10 7.93888733e-10]
Número de iteraciones: 1
Método del Punto Proximal:
```

	Iteración	Iteraciones internas	x0	f(x0)	||x0 - x||
0	1	1	[7.938887325309452e-10, 7.938887325309452e-10]	-1.0	1.122728e-09