Tarefa-02 - MO431

Patrick de Carvalho Tavares Rezende Ferreira - 175480

Neste experimento foram executadas operações para localização de mínimos locais na função de Rosenbrock 3D. Foi utilizada a formulação do gradiente explícito e do gradiente com auxílio da API do tensorflow. Ambas as abordagens produziram resultados praticamente idênticos e são descritas ao final deste procedimento.

Abaixo são importadas as bibliotecas e pacotes utilizados neste roteiro.

In [69]:

```
from numpy.linalg import norm
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

%matplotlib inline
```

Abaixo é definida a função de rosenbrock 3d. Ela recebe o trio de ordenadas (x1, x2, x3) como um array ou lista e retorna o valor da função naquele ponto.

In [70]:

```
def rosenbrock_3d(x):
    """
Funcao de Rosenbrock em 3d
    :param x: Array de ordenadas da funcao (x1, x2, x3).
    :return: Retorna o valor da funcao no ponto (x[0], x[1], x[2]).
    """
    x = np.array(x)

    return 100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0]) ** 2 + 100 * (x[2] - x[1]
** 2) ** 2 + (1 - x[1]) ** 2
```

O roteiro deste experimento solicitou que o critério de parada da otimização do gradiente fosse a tolerância da variação entre os pontos da função, sendo definida como a norma da diferença entre o ponto anterior e o atual, dividida pela norma do ponto anterior. A função que faz este cômputo está definida abaixo.

In [71]:

```
def tolerancia(x, x_old):
    """
Computa a tolerancia para o gradiente. Serve para indicar quando o gradiente
esta proximo do minimo e ja esta parando de deslocar o ponto indicado.
    :param x: ponto atual gerado pelo gradiente
    :param x_old: ponto anterior do caminho sendo percorrido pelo gradiente
    :return: Razao entre a variacao do ponto atual para o anterior.
    """
    x = np.array(x)
    x_old = np.array(x_old)

    return norm(x - x_old) / norm(x_old)
```

A função abaixo computa numericamente o gradiente para a função de rosenbrock 3d num dado ponto "x" (x1, x2, x3).

In [72]:

```
def gradiente_rosenbrock_3d(x, step_derivativo=10 ** -10):
    """
Computa o gradiente da funcao rosenbrock 3d
    :param x: Ponto a avaliar o gradiente.
    :param step_derivativo: Largura do intervalo a calcular as derivadas parciai
s
    :return: Gradiente avaliado no ponto x (x1, x2, x3).
    """
    x = np.array(x)

    df1 = (rosenbrock_3d(x + np.array([step_derivativo, 0, 0])) - rosenbrock_3d(
x)) / step_derivativo
    df2 = (rosenbrock_3d(x + np.array([0, step_derivativo, 0])) - rosenbrock_3d(
x)) / step_derivativo
    df3 = (rosenbrock_3d(x + np.array([0, 0, step_derivativo])) - rosenbrock_3d(
x)) / step_derivativo

    return np.array([df1, df2, df3])
```

Abaixo temos a função que executa o script de minimização através da formulação explícita do gradiente convencional (sem momento). O passo-a-passo é descrito em detalhes em seus comentários.

In [73]:

```
def minimizacao_gradiente_explicito(lr=10 ** -3, max_passos=20000):
    """
Realiza a minimizacao por gradiente convencional (SGD sem momento) e plota os
resultados.
    :param lr: Learning Rate (deafult: 10 ** -3).
    :param max_passos: Limite de iteracoes para minimizacao (default: 20000).
    """

    # Ponto inicial da fncao onde calcularemos o gradiente
    x = np.array([0, 0, 0])
    # Para calcular a variacao de um ponto "x" em relacao ao anterior, devemos
    # salvar o anteriror.
    x_old = np.copy(x)
    # Uma lista para guardar os valores de cada iteracao e poder plotar os
    # resultado ao final.
    plot_rosenbrock = list()
    # Uma lista para guardar os valores de "x" em cada iteracao e poder plotar
    # os resultado ao final.
    plot_x = list()

    # A funcao de tolerancia daria erro na primeira iteracao por estarmos
    # sobre o ponto zero. Fazemos portanto a primeira iteracao fora do loop e
    # sem calcular a tolerancia.
    plot_rosenbrock.append(rosenbrock_3d(x))
    plot_x.append(np.copy(x))
    x = x - lr * gradiente_rosenbrock_3d(x)

    # Neste loop repetimos a sequencia de salvar o ponto anterior, calcular o
    # gradiente, usa-lo para calcular o novo ponto e verificar o criterio de
    # tolerancia. Temos um limite de iteracoes em "max_passos", caso o gradiente
    # comece a divergir, mas a intencao eh que o limite de tolerancia seja
    # batido e a instrucao "break" rompa o loop.
    for i in range(max_passos):
        # Salva ponto anterior
        x_old = np.copy(x)

        # Salva os dados para posterior plotagem
        plot_rosenbrock.append(rosenbrock_3d(x))
        plot_x.append(np.copy(x))
        # Usa o gradiente para plotar os novos pontos.
        x = x - lr * gradiente_rosenbrock_3d(x)

        # Verifica o criterio de tolerancia
        if tolerancia(x, x_old) < 10 ** -4:
            # Se ja estamos abaixo da tolerancia, salva o ultimo dado e encerra.
            plot_rosenbrock.append(rosenbrock_3d(x))
            break

    # Plotamos os valores de f(x) para cada atualizacao de "x".
    plt.title("Valores para cálculo do gradiente explícito com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("f(x) - Rosenbrock 3d")
    plt.plot(plot_rosenbrock)
    plt.show()

    # Exibimos na tela as informacoes de interesse.
    print("\nNúmero de passos do gradiente: ", len(plot_rosenbrock))
    print("\nValor da função no mínimo local: ", rosenbrock_3d(x))
```

```
    print("Valores de x1: ", x[0], ", x2: ", x[1], "e x3: ", x[2])

    # Tranformamos em uma matriz numpy para poder usar a API de indices.
    plot_x = np.array(plot_x)

    # Abaixo plotamos os valores das ordenadas "x" para entendermos como as
    # "hipoteses" do gradiente se comportam.
    plt.title("Valores para cálculo do gradiente explícito com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x1")
    plt.plot(plot_x[:, 0].reshape(plot_x[:, 0].shape[0]))
    plt.show()

    plt.title("Valores para cálculo do gradiente explícito com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x2")
    plt.plot(plot_x[:, 1].reshape(plot_x[:, 0].shape[0]))
    plt.show()

    plt.title("Valores para cálculo do gradiente explícito com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x3")
    plt.plot(plot_x[:, 2].reshape(plot_x[:, 0].shape[0]))
    plt.show()
```

Abaixo temos a função que executa o script de minimização através da API do tensorflow com gradiente convencional (sem momento). O passo-a-passo é descrito em detalhes em seus comentários.

In [74]:

```
def minimizacao_gradiente_tensorflow(lr=10 ** -3, max_passos=20000):
    """
Realiza a minimizacao por gradiente convencional (SGD sem momento) usando a API
do tensorflow.
    :param lr: Learning Rate (deafult: 10 ** -3).
    :param max_passos: Maximo de iteracoes a realizar para a otimizacao (defaul
t: 20000).
    """

    # Para calcular a tolerancia, temos que saber quanto os pontos de ordenadas
    # "x" valiam antes. Para isto que servem estas variaveis.
    x1_old, x2_old, x3_old = tf.Variable(0), tf.Variable(0), tf.Variable(0)

    # Precisam ser variaveis do tensorflow para o graidiente poder alterar o
    # valor da funcao e computar as derivadas.
    x1, x2, x3 = tf.Variable(10 ** -10), tf.Variable(10 ** -10), tf.Variable(10
** -10)

    # Selecionamos o tipo de otimizador, que sera o gradiente comum, uma vez que
    # o SGD sem momento iguala a um gradiente simples.
    opt = tf.keras.optimizers.SGD(lr=lr)

    # Salvamos aqui os valores de "x", doas gradientes e da propria funcao f(x)
    # para pode plotar e analisar apos a otimizacao.
    lista_plot = list()

    # Se a iteracao chegar ao numero "max_passos", significa que o calculo
    # divergiu e devemos parar. A intencao eh que a tolerancia de minimizacao
    # seja atingida antes e a instrucao break seja chamada.
    for i in range(max_passos):
        # A funcao escolhida para avaliar as derivadas eh o GradientTape
        with tf.GradientTape() as tape:
            # A funcao sobre a qual calculamos os gradientes (rosenbrock 3d).
            y = 100 * (x2 - x1 ** 2) ** 2 + (1 - x1) ** 2 + 100 * (x3 - x2 ** 2)
** 2 + (1 - x2) ** 2
            # Obtemos os valores dos gradientes no ponto "x".
            grads = tape.gradient(y, [x1, x2, x3])
            # Colocamos eles em uma nova lista.
            processed_grads = [g for g in grads]
            # Zipamos os gradientes e os valores de "x" juntos para computar os
            # proximos valores de "x".
            grads_and_vars = zip(processed_grads, [x1, x2, x3])

        # Coloamos na lista de registros de dados para o grafico os valores
        # desta iteracao.
        lista_plot.append([y.numpy(),
                           x1.numpy(),
                           x2.numpy(),
                           x3.numpy(),
                           grads[0].numpy(),
                           grads[1].numpy(),
                           grads[2].numpy()]
                          )

        # Atualizamos as variaveis que guardam a iteracao anterior para poder
        # gerar o novo ponto "x".
        x1_old, x2_old, x3_old = x1.numpy(), x2.numpy(), x3.numpy()
        # Obtemos os novos valores de "x" atraves do otimizador escolhido (SGD).
        opt.apply_gradients(grads_and_vars)
```

```
        # Se a diferenca relativaentre o novo ponto gerado e o antigo for menor
        # que o especificado, estamos proximos o suficiente do minimo e paramos
        # aqui.
        if tolerancia([x1, x2, x3], [x1_old, x2_old, x3_old]) < 10 ** -4:
            break

    # Tranformamos a lista de registros em numpy array para poder usar sua API.
    lista_plot = np.array(lista_plot)

    # Exibimos o grafico de saida com os valores da funcao a cada atualizacao de
    # "x".
    plt.title("Valores para cálculo do gradiente tensorflow com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("f(x) - Rosenbrock 3d")
    plt.plot(lista_plot[:, 0].reshape(lista_plot[:, 0].shape[0]))
    plt.show()

    # Informamos os valores de interesse. 1 passo somado a mais devido ao
    # deslocamento do ponto zero que fazemos incialmente para resolver a
    # singularidade da regiao.
    print("\nNúmero de passos do gradiente: ", lista_plot.shape[0] + 1)
    print("\nValor da função no mínimo local: ", rosenbrock_3d([x1.numpy(), x2.n
umpy(), x3.numpy()]))
    print("\nValores de x1: ", x1.numpy(), ", x2: ", x2.numpy(), "e x3: ", x3.nu
mpy())

    # Abaixo plotamos os valores das ordenadas "x" para entendermos como as
    # "hipoteses" do gradiente se comportam.
    plt.title("Valores para cálculo do gradiente tensorflow com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x1")
    plt.plot(lista_plot[:, 1].reshape(lista_plot[:, 0].shape[0]))
    plt.show()

    plt.title("Valores para cálculo do gradiente tensorflow com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x2")
    plt.plot(lista_plot[:, 2].reshape(lista_plot[:, 0].shape[0]))
    plt.show()

    plt.title("Valores para cálculo do gradiente tensorflow com LR de " + str(lr
))
    plt.xlabel("Número de atualizações de x")
    plt.ylabel("Valor de x3")
    plt.plot(lista_plot[:, 3].reshape(lista_plot[:, 0].shape[0]))
    plt.show()
```

1 Implementação de descida do gradiente com gradiente explicito

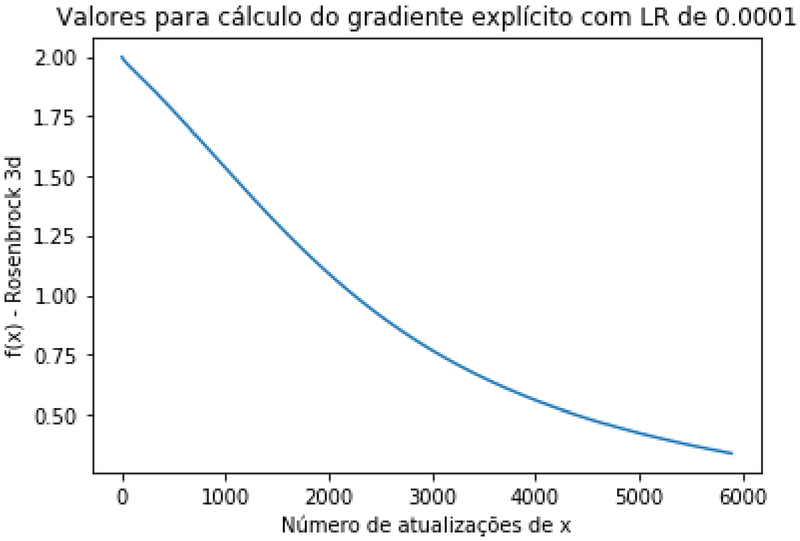
A partir de agora executamos as descidas do gradiente com gradiente explícito. As funções que calculam o gradiente numericamente e a tolerância foram citadas e explciadas nas células acima.

Na célula abaixo, fazemo a descida do gradiente com learning rate de 10^-4. O primeiro gráfico exibe o valor da função a cada atualização do vetor "x", e logo abaixo é informado que a função convergiu em 5894 passos, quando atingiu o critério de tolerância e obteve o valor de 0.3387147320704825. Os valores das ordenadas foram de x1: 0.7063967177312058 , x2: 0.4983415713977202 e x3: 0.24550214694496475.

Nos 3 últimos gráficos da célula abaixo são exibidos os progressos dos valores das ordenadas (x1, x2 e x3) a cada atualização, apenas para visualização de como sua mudança de valores se dá.

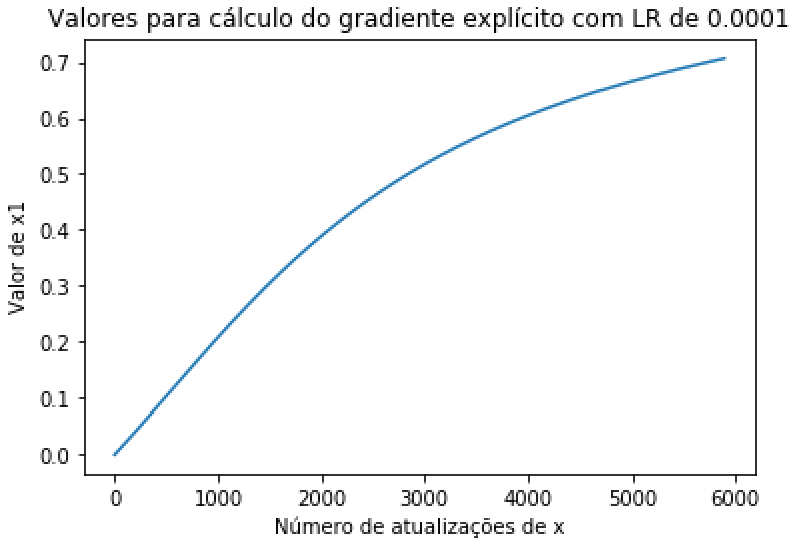
In [75]:

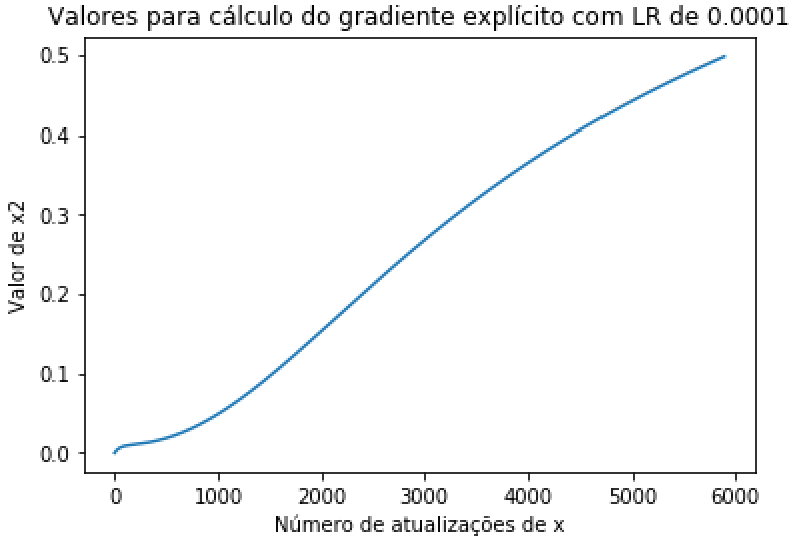
```
# Learning rate pequeno aumenta a chance de convergir, ao passo que diminui
# a velocidade em que se alcanca o minimo local.
minimizacao_gradiente_explicito(lr=1.00 * 10 ** -4)
```

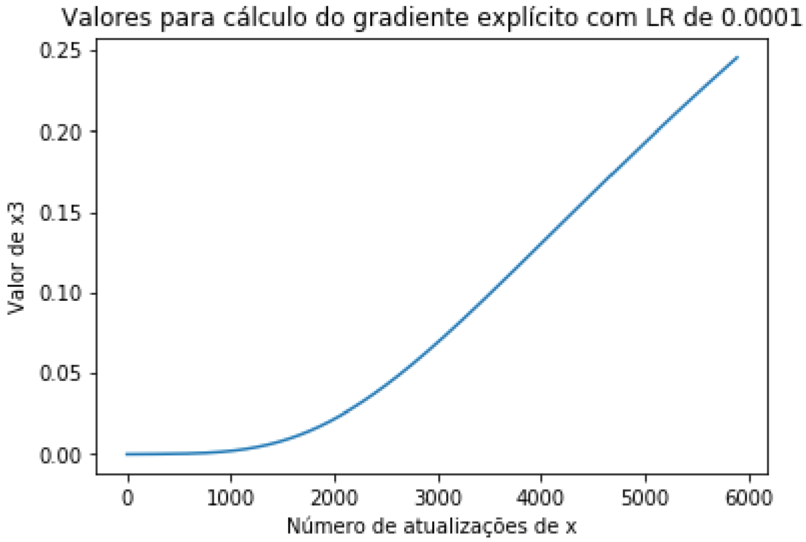



```
Número de passos do gradiente:  5894

Valor da função no mínimo local:  0.3387147320704825
Valores de x1:  0.7063967177312058 , x2:  0.4983415713977202 e x3:
0.24550214694496475
```





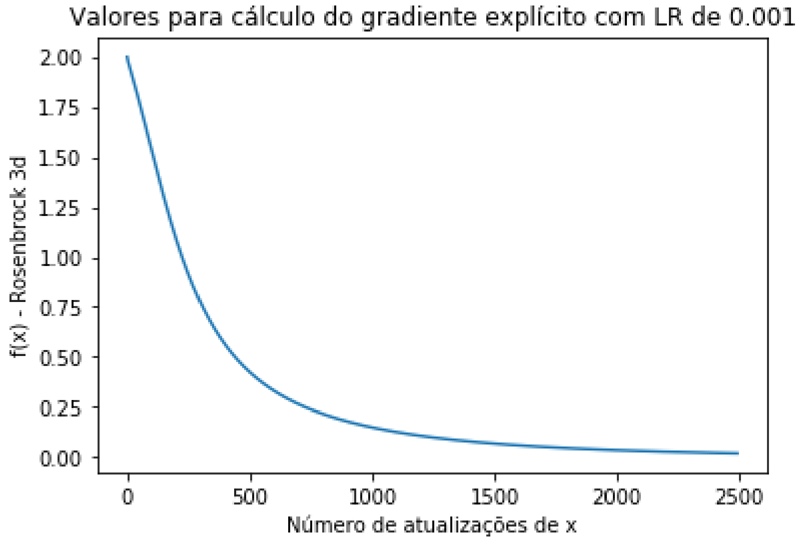


Na célula abaixo, fazemos a descida do gradiente com learning rate de 10^-3, que tende a nos levar para o mínimo local em menor tempo do que o learning rate pequeno do item anterior. O primeiro gráfico exibe o valor da função a cada atualização do vetor "x", e logo abaixo é informado que a função convergiu em 2492 passos, quando atingiu o critério de tolerância e convergiu para o valor de 0.019028770294624527. Os valores das ordenadas foram de x1: 0.7063967177312058 , x2: 0.4983415713977202 e x3: 0.24550214694496475.

Nos 3 últimos gráficos da célula abaixo são exibidos os progressos dos valores das ordenadas (x1, x2 e x3) a cada atualização, apenas para visualização de como sua mudança de valores se dá.

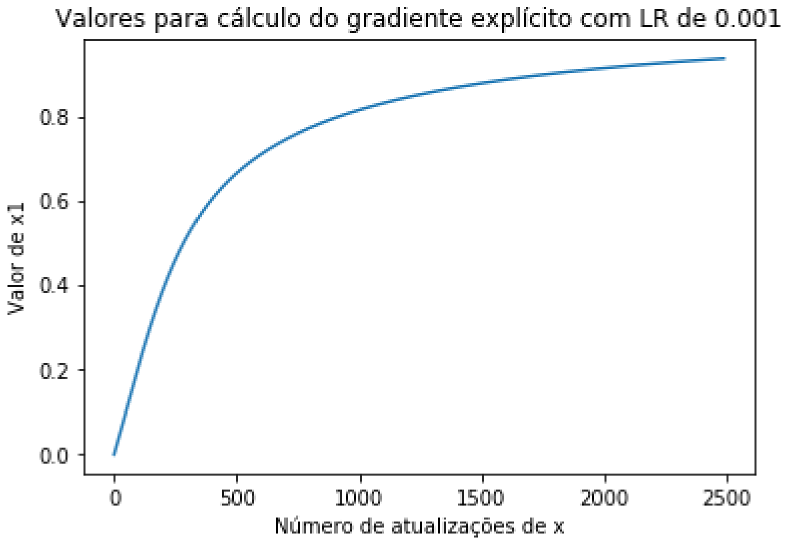
In [76]:

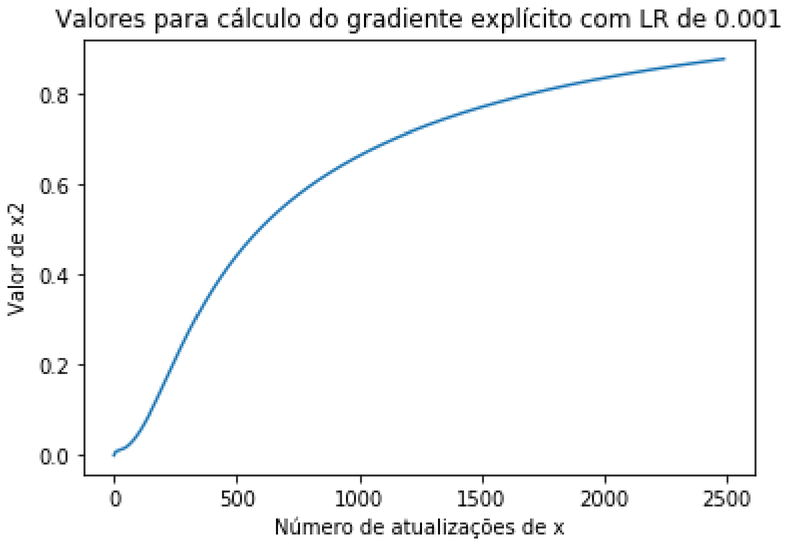
```
# Learning rate um pouco mais alto permite diminuir o tempo de convergencia.
minimizacao_gradiente_explicito(lr=1.00 * 10 ** -3)
```

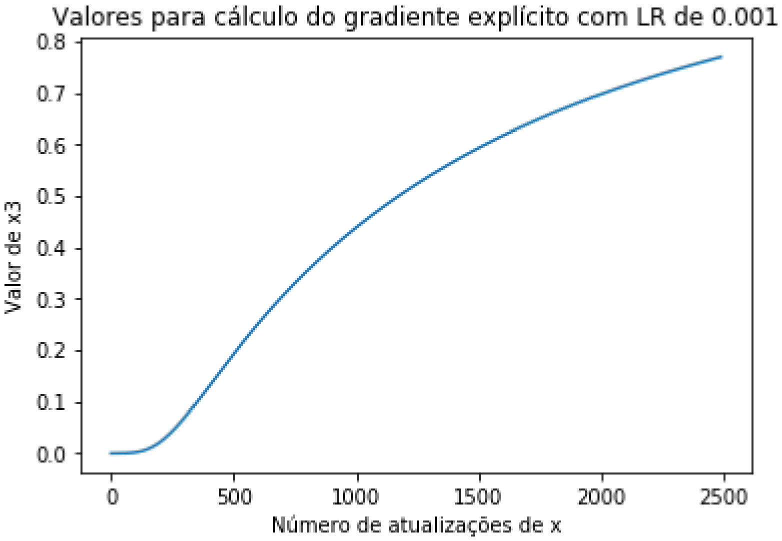



```
Número de passos do gradiente:  2492

Valor da função no mínimo local:  0.019028770294624527
Valores de x1:  0.9368876938489756 , x2:  0.8775232136770672 e x3:
0.7694184560660955
```







Na célula abaixo, fazemos a descida do gradiente com learning rate de 1.62*10^-3, que é um valor alto demais e faz com que o gradiente passa direto pelos mínimos locais da função e entre numa região de divergência da função, parando somente quando o limite de passos é extrapolado. O primeiro gráfico exibe o valor da função a cada atualização do vetor "x", e logo abaixo é informado que a função obtinha quando ultrapassou o limite de 20000 passos, tendo o valor de 0.24522489117768181. Os valores das ordenadas foram de 0.9329722032342531 , x2: 0.906341867032423 e x3: 0.7893549070381312, onde nota-se a proximidade dos mínimos locais obtidos anteriormente e como a região de convergência e divergência podem ser próximas.

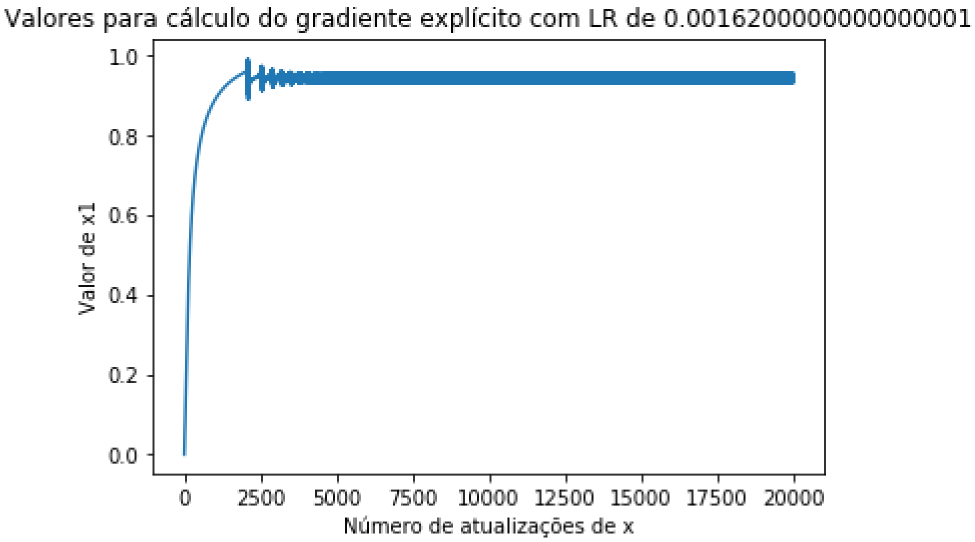
Nos 3 últimos gráficos da célula abaixo são exibidos os progressos dos valores das ordenadas (x1, x2 e x3) a cada atualização, apenas para visualização de como sua mudança de valores se dá.

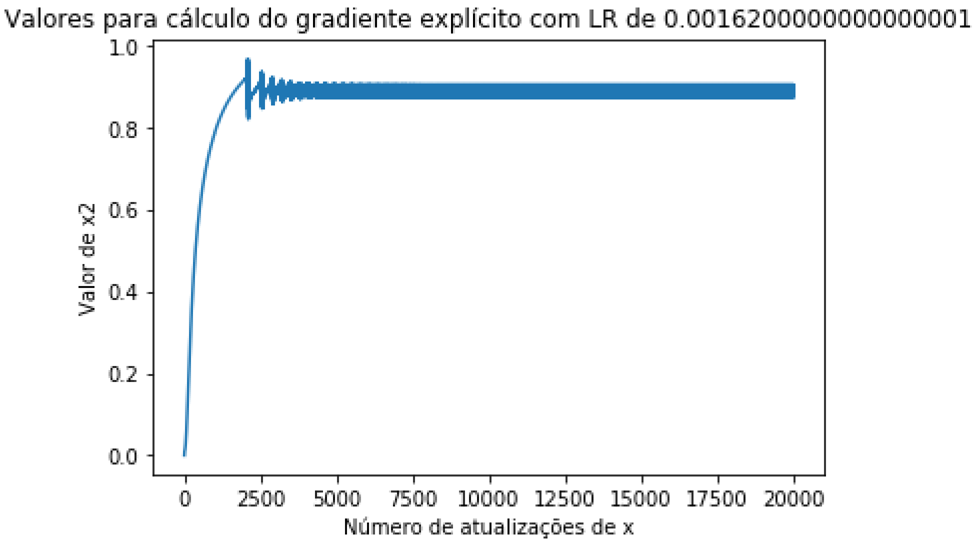
In [77]:

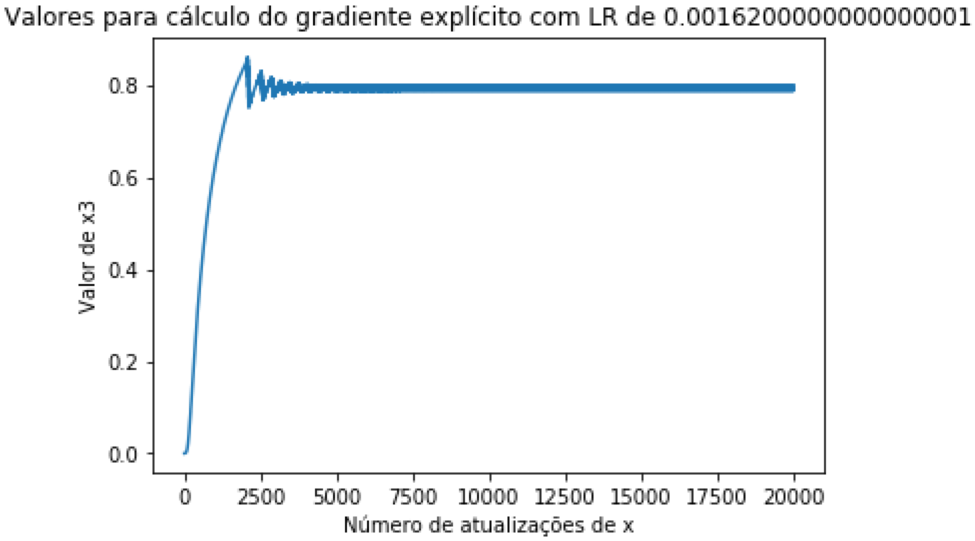
```
# Learning rate alto demais causando a extrapolacao do minimo local e a
# divergencia do gradiente.
minimizacao_gradiente_explicito(lr=1.62 * 10 ** -3)
```


```
Número de passos do gradiente:  20001

Valor da função no mínimo local:  0.24522489117768181
Valores de x1:  0.9329722032342531 , x2:  0.906341867032423 e x3:
0.7893549070381312
```







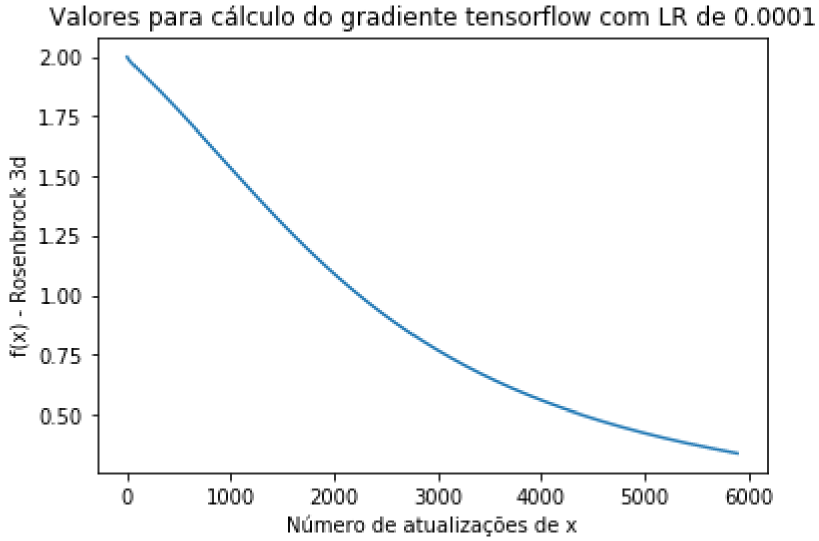
2 Usando do tensorflow para calcular o gradiente

Na célula abaixo, fazemos a descida do gradiente com learning rate de 10^-4 e usando a API do tensorflow para implementar o gradiente sem momento. O primeiro gráfico exibe o valor da função a cada atualização do vetor "x", e logo abaixo é informado que a função convergiu em 2492 passos, quando atingiu o critério de tolerância e convergiu para o valor de 0.3387153160218814. Os valores das ordenadas foram de x1: 0.7063964 , x2: 0.49834117 e x3: 0.24550174. Nota-se que a função convergiu com exatamente a mesma quantidade de iterações no caso de mesmo learning rate implementado manualmente e encontrou o mesmo mínimo local, indicando que as metodologias empregadas são coerentes entre si.

Nos 3 últimos gráficos da célula abaixo são exibidos os progressos dos valores das ordenadas (x1, x2 e x3) a cada atualização, apenas para visualização de como sua mudança de valores se dá.

In [78]:

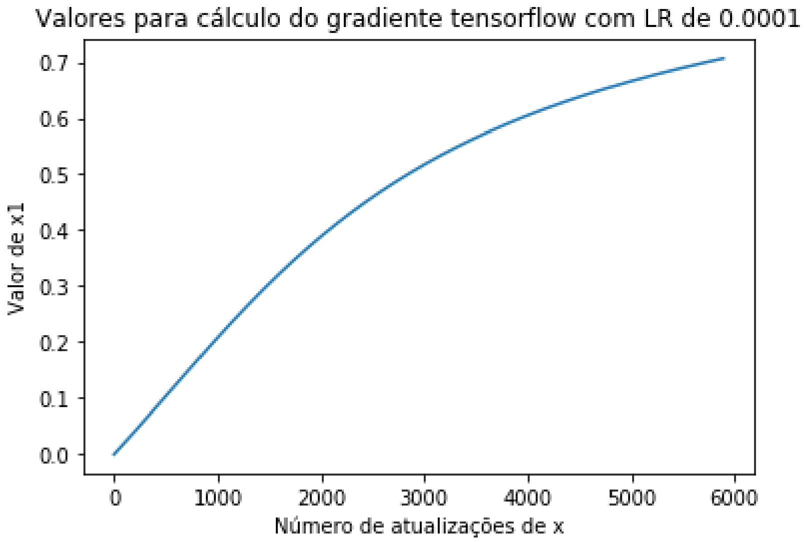
```
minimizacao_gradiente_tensorflow(lr=1.00 * 10 ** -4)
```

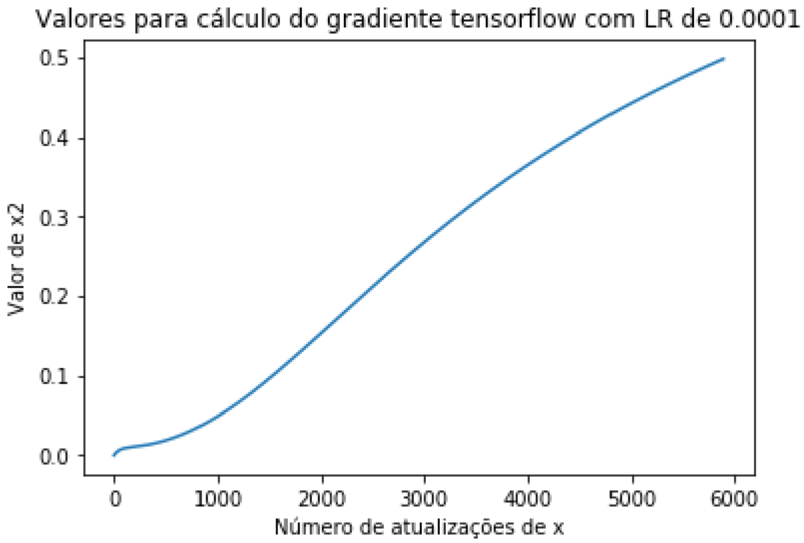


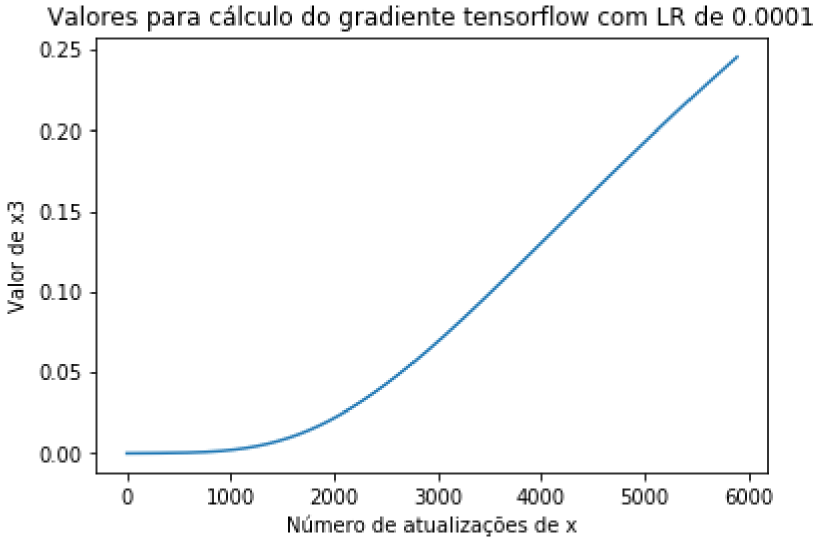

```
Número de passos do gradiente:  5894

Valor da função no mínimo local:  0.3387153160218814

Valores de x1:  0.7063964 , x2:  0.49834117 e x3:  0.24550174
```





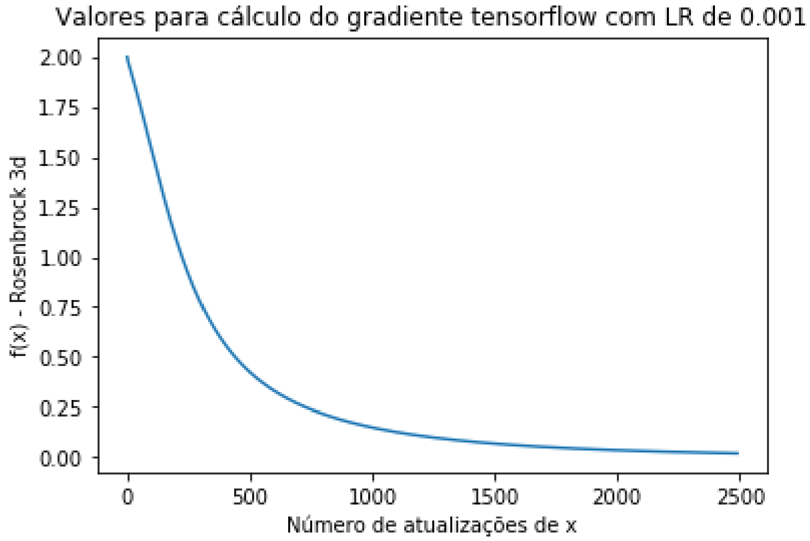


Na célula abaixo, fazemos a descida do gradiente com learning rate de 10^-3, que tende a nos levar para o mínimo local em menor tempo do que o learning rate pequeno do item anterior, e usando a API do tensorflow para implementar o gradiente sem momento. O primeiro gráfico exibe o valor da função a cada atualização do vetor "x", e logo abaixo é informado que a função convergiu em 2492 passos, quando atingiu o critério de tolerância e convergiu para o valor de 0.01902871111779364. Os valores das ordenadas foram de x1: 0.9368878 , x2: 0.8775234 e x3: 0.7694188. Nota-se que a função convergiu com exatamente a mesma quantidade de iterações no caso de mesmo learning rate implementado manualmente e encontrou o mesmo mínimo local, indicando que as metodologias empregadas são coerentes entre si.

Nos 3 últimos gráficos da célula abaixo são exibidos os progressos dos valores das ordenadas (x1, x2 e x3) a cada atualização, apenas para visualização de como sua mudança de valores se dá.

In [79]:

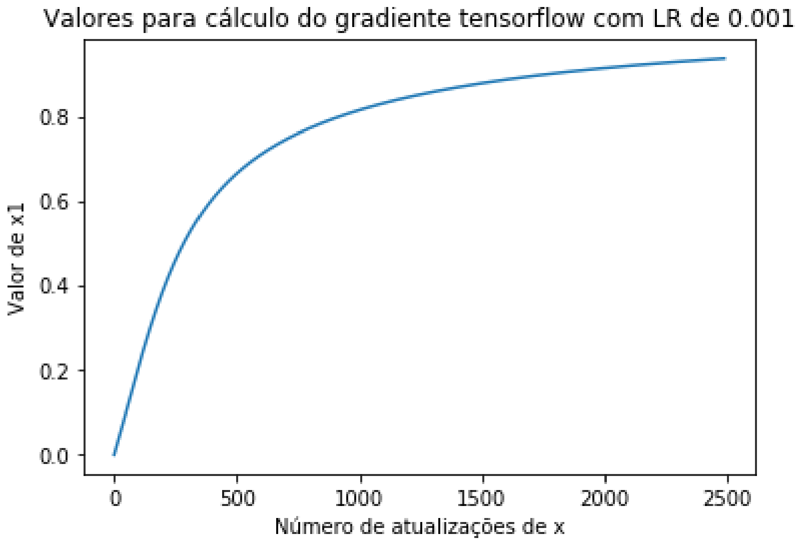
```
minimizacao_gradiente_tensorflow(lr=1.00 * 10 ** -3)
```

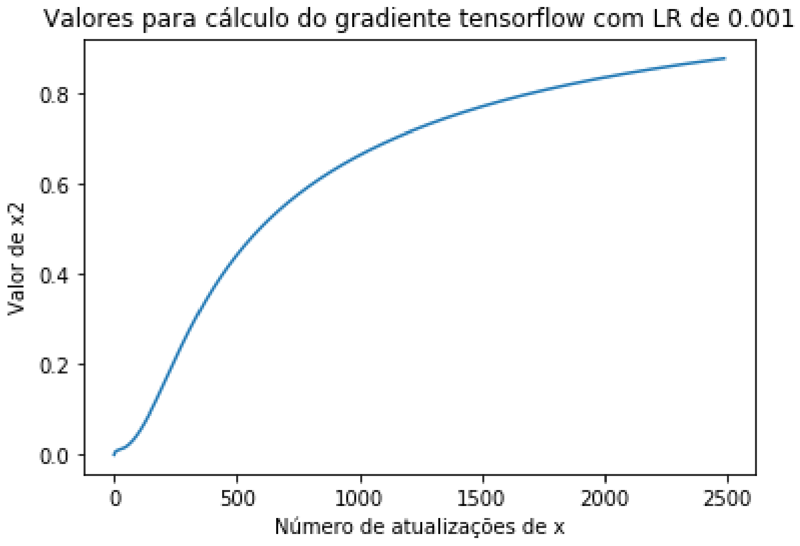


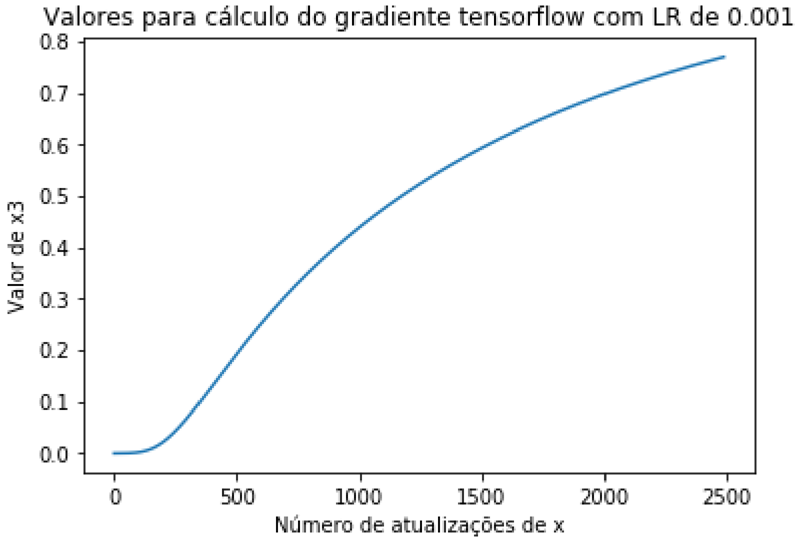

```
Número de passos do gradiente:  2492

Valor da função no mínimo local:  0.019028711111779364

Valores de x1:  0.9368878 , x2:  0.8775234 e x3:  0.7694188
```






Valores para cálculo do gradiente tensorflow com LR de 0.001
Valor de x3
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
500
1000
1500
2000
2500
Número de atualizações de x