Курсова робота

із дисципліни «Методи оптимізації»

на тему: Метод Девідона-Флетчера-Пауелла

Студента групи КМ-82

Буслаєва В. О.

Керівник:

Викладач Норкін Б. В.

Зміст

Вступ

Теорія оптимізації знаходить ефективне застосування в усіх напрямках інженерної діяльності, і в першу чергу в чотирьох її областях:

1) проектування систем і їх складових частин 2) планування та аналіз функціонування існуючих систем 3) інженерний аналіз і обробка інформації 4) управління динамічними системами.

Теорія оптимізації являє собою сукупність фундаментальних математичних результатів і чисельних методів, орієнтованих на знаходження і ідентифікацію найкращих варіантів з безлічі альтернатив і дозволяють уникнути повного перебору і оцінювання можливих варіантів.

В даній курсовій роботі нами буде розглянуто алгоритм мінімізації функції Розенброка методом Девідона-Флетчера-Пауелла. В ході роботи буде розглянуто умовну оптимізацію, а також варіанти, коли глобальний мінімум знаходиться всередені та поза допутимою областю. Для обчислення буде використано метод внутрішньої точки.

Матеріали та методи.

Як відомо, задача умовної оптимізації методом штрафних функцій зводиться до послідовності вирішення задач безумовної оптимізації. В таких методах використовується принцип штрафу, що додається до початкової функції, та значення якого збільшується з кожним разом.

В даній роботі було використано метод Девідона-Флетчера-Пауелла, що відноситься до квазіньютонівських методів.

Для будь-якої задачі безумовної оптимізації можна використовувати метод Ньютона, так як для нього доведена збіжність. Проте, для застосування методу Ньютона треба знати матрицю Гессе даної функції і здійснити зворотнє перетворення цієї матриці. У багатьох випадках може виявитися, що матриці Гессе невідомі, або їх обчислення пов'язано з великими труднощами, або вони можуть бути отримані тільки чисельними методами. Методи змінної метрики апроксимують матрицю Гессе або зворотню до неї, використовуючи для цього значення тільки перших похідних. У більшості цих методів використовуються спряжані напрями. Перехід з точки $x^{(k)}$ в точку $x^{(k+1)}$ визначається наступним чином:

$x^{(k+1)} = x^{(k)} - \lambda^{*(k)}A(x^{(k)})\nabla f(x)$

де

$A(x^{(k)})$ - матриця порядку nXn, називається метрикою, змінюється на кожній ітерації і являє собою апроксимацію оберненої матриці Гессе;

$\lambda^{*(k)}$ - скаляр, який визначається за допомогою методів одновимірного пошуку;

$\nabla f(x)$ - значення градієнта в точці $x^{(k)})$

Метод Девідона - Флетчера - Пауелла не вимагає обчислення зворотної матриці Гессе. Матриця напрямків A обчислюється таким чином, щоб для квадратичної цільової функції в межі після n кроків вона дорівнювала H^{-1} . Вихідна матриця A зазвичай вибирається у вигляді одиничної матриці $A^{(0)} = I$ (але може бути і будь-якою симетричною додатньо визначеною матрицею), так що вихідний напрямок мінімізації - це напрямок найшвидшого спуску. В ході оптимізації відбувається поступовий перехід від напрямку найшвидшого

спуску до ньютоновскому (на відповідних етапах мінімізації використовуються переваги кожного з цих підходів).

Алгоритм методу Девідона - Флетчера - Пауелла.

Вихідні дані:

- Початкова точка $x^{(0)}$;
- Параметри закінчення $\epsilon_{1}, \epsilon_{2}$.

1) Задати координати початкової точки $x^{(0)}$. Обчислити значення $\nabla f(x)$. Покласти $A^{(0)} = I$. 2) Обчислити $\lambda^{*(k)}$ при мінімізації функції $f\lfloor x^{(k)} - \lambda^{*(k)}A(x^{(k)})\nabla f(x) \rfloor$ (Використовується одновимірний метод пошуку). 3) Обчислити точку $x^{(k+1)}$:

$x^{(k+1)} = x^{(k)} - \lambda^{*(k)}A(x^{(k)})\nabla f(x)$

4) Обчислити $f(x^{(k+1)}), \nabla f(x^{(k+1)})$,

$\Delta g(x^{(k)})=\nabla f(x^{(k+1)})-\nabla f(x^{(k)}),\Delta x^{(k)}=x^{(k+1)}-x^{(k)}$.

5) Перевірити критерій закінчення 6) Якщо критерій закінчення виконався, пошук закінчено. Інакше повернутися до п. 2. 7) k = k+1. Обчислити $A(x^{(k)})$

$$A^{(k)} = A^{(k-1)} + \frac{\Delta x^{(k-1)} \Delta x^{(k-1)^T}}{\Delta x^{(k-1)^T} \Delta g^{(k-1)}} - \frac{A^{(k-1)} \Delta g^{(k-1)} \Delta g^{(k-1)^T} A^{(k-1)}}{\Delta g^{(k-1)^T} A^{(k-1)} \Delta g^{(k-1)}}.$$

Ця формула зберігає властивість симетрії і позитивної визначеності матриць. Таким чином, алгоритм забезпечує спадання цільової функції від ітерації до ітерації.

У методі Девідона-Флетчера-Пауелла можуть зустрічатися негативні кроки (накопичення помилки). Для уникнення цього:

1) Збільшити точність. 2) Продовжити пошук з переглянутою матрицею напрямків.

Для вирішення задачі умовної оптимізації було обрано метод внутрішньої точки.

Методи внутрішньої точки використовують такі штрафи Ф, при яких стаціонарні точки штрафний функції P (x, R) є допустимими. Штраф створює уздовж кордону допустимої області бар'єр з нескінченно великих значень функції P (x, R). Ці методи називають методами бар'єрів. Пошук мінімуму починається з точки, в якій всі обмеження виконуються як строгі нерівності. При мінімізації штрафний функції вихід на межу допустимої області, де штраф Ф нескінченний, не має сенсу, тому процес руху до мінімуму функції ніколи не покине допустиму область.

Вирішення проблеми

Імпорт бібліотек

In [1]:

```
# Import the libraries we need to use
import numpy as np
import math
import matplotlib.pyplot as plt
```

Визначення констант

In [2]:
```
LAMBDA0 = 0                         # starting lambda
COUNTER = 0                         # counter of calculating of function
SVEN_PAR = 0.35                     # parametr that we place in delta_lambda function
EPSILON = 0.01                      # precision of calculating length of step
PRECISE = 0.001                      # precision of criterion of stop
H = 0.001                           # step in numerical calculating of derivative
R = 1                               # multiplier for penalty function
A0 = np.matrix([[1, 0], [0, 1]])    # starting A matrix
X0 = (-1.2, 0)                      # starting dot
```

Визначення функцій

In [3]:
```
# help function to check the error meaning
def penalty(x1, x2):
        try:
                log = math.log(g(x1, x2))
        except:
                log = -1000
        return -log
```

In [4]:
```
# this is function with equation which define the possible area
def g(x1, x2):
    # we will firstly try a situation, when both start and minimal dots are in the a
    # that's a simulation of optimization without limitations
        return 25-(x1+1)**2-x2**2
```

In [5]:
```
# this is the main function with calculation of Rosenbroks' function with penalty
def f(x1, x2):
        global COUNTER
        COUNTER += 1
        foo = (1-x1)**2 + 100*((x2-x1**2)**2) + R*penalty(x1, x2)
        return foo
```

In [6]:
```
# this function numerically counts meaning of gradient of the function in the dot
def grad(x1, x2):
        deriv1 = (f(x1+H, x2) - f(x1-H, x2))/(2*H)
        deriv2 = (f(x1, x2+H) - f(x1, x2-H))/(2*H)
        return (deriv1, deriv2)
```

In [7]:
```
# additional functions which just calculate norm of some dot and dalta in counting l
def norm(x1, x2):
        return math.sqrt(x1**2 + x2**2)

def delta_lambda(x1, x2, s1, s2):
        return SVEN_PAR*(norm(x1, x2)/norm(s1, s2))
```

In [8]:

```
# while calculating lambda with Svenn algorithm there might appear such situation, w
# this is where we use this function using numerical approximation of that meaning
def sven_approximate(lambda_in, lambda_out, x1, x2, s):
        lambdas = [(lambda_in, True), (lambda_out, False)]
        while not (lambdas[-1][1] == False and lambdas[-2][1] == False):

                true_lambda, false_lambda = 0, 0
                for el in lambdas:
                        if el[1]:
                                true_lambda = el[0]
                        else:
                                false_lambda = el[0]

                new_lambda = (true_lambda + false_lambda)/2
                if g(x1+s[0]*new_lambda, x2+s[1]*new_lambda) < 0:
                        lambdas.append((new_lambda, False))
                else:
                        lambdas.append((new_lambda, True))

                if len(lambdas) > 100:
                        for el in lambdas[::-1]:
                                if el[1]:
                                        return el[0]

        return lambdas[-3][0]
```

In [9]:

```
# this is function which realizes Svenn algorithm
# it returns an interval where the minimum of f(x) lies
# here we transform our function with two parametrs in the function with one paramet
def alg_sven(x1, x2, delta, lambda0, s):
        lambdas = [lambda0]
        f_0 = f(x1+s[0]*lambda0, x2+s[1]*lambda0)

        lambda_left = lambda0 - delta
        f_left = f(x1+s[0]*lambda_left, x2+s[1]*lambda_left)

        lambda_right = lambda0 + delta
        f_right = f(x1+s[0]*lambda_right, x2+s[1]*lambda_right)

        if f_left < f_0:
                mode = 'left'
                lambdas.append(lambda_left)
        else:
                mode = 'right'
                lambdas.append(lambda_right)

        _continue = True
        out_lambda = 0
        while f(x1+s[0]*lambdas[-1], x2+s[1]*lambdas[-1]) < f(x1+s[0]*lambdas[-2], x

                if mode == 'left':
                        new_lambda = lambdas[-1] - delta*(2**(len(lambdas)-1))
                else:
                        new_lambda = lambdas[-1] + delta*(2**(len(lambdas)-1))

                if g(x1+s[0]*new_lambda, x2+s[1]*new_lambda) >= 0:
                        lambdas.append(new_lambda)
                else:
                        _continue = False
                        out_lambda = new_lambda

        if out_lambda == 0:
                new_lambda = (lambdas[-1]+lambdas[-2])/2
                if f(x1+s[0]*new_lambda, x2+s[1]*new_lambda) < f(x1+s[0]*lambdas[-2]
```

```
                        a, b = lambdas[-2], lambdas[-1]
                else:
                        idx = -2 if len(lambdas)<3 else -3
                        a, b = lambdas[idx], new_lambda

        else:
                new_lambda = (lambdas[-1]+out_lambda)/2
                while g(x1+s[0]*new_lambda, x2+s[1]*new_lambda) < 0:
                        new_lambda = (lambdas[-1]+new_lambda)/2

                if f(x1+s[0]*new_lambda, x2+s[1]*new_lambda) > f(x1+s[0]*lambdas[-1]
                        if f(x1+s[0]*new_lambda, x2+s[1]*new_lambda) < f(x1+s[0]*lam
                                a, b = lambdas[-1], new_lambda
                        else:
                                a, b = lambdas[-1], lambdas[-2]

                else:
                        lambda_approx = sven_approximate(new_lambda, out_lambda, x1,
                        a, b = new_lambda, lambda_approx

        return min(a,b), max(a,b)
```

In [10]:

```
# to minimize lambda we will use method of golden cut
# it is also possible to use DSK-Pauell method yet because of numerical counts there
def golden(x1, x2, s, a, b, epsilon):
        L = abs(b-a)
        while L > epsilon:
                x1_gold = a + 0.382*L
                x2_gold = a + 0.618*L

                f_x1 = f(x1+s[0]*x1_gold, x2+s[1]*x1_gold)
                f_x2 = f(x1+s[0]*x2_gold, x2+s[1]*x2_gold)

                center = min(f_x1, f_x2)
                if center == f_x1:
                        b = x2_gold
                else:
                        a = x1_gold
                L = abs(b-a)

        return (a + b)/2
```

In [11]:

```
# those are additional functions to count deltas
def x_delta(X1, X2):
        return X2[0]-X1[0], X2[1]-X1[1]

def g_delta(X1, X2):
        g1 = grad(X1[0], X1[1])
        g2 = grad(X2[0], X2[1])
        return g2[0]-g1[0], g2[1]-g1[1]
```

In [12]:

```
# this function counts new way using A-matrix and gradient
def count_s(x1, x2, A):
        gradient = np.matrix(grad(x1, x2))
        S = -A@gradient.T
        return S.item(0), S.item(1)
```

In [13]:

```
# this function counts next A-matrix
def count_A(A, X1, X2):
```

```
        g_del = np.matrix(g_delta(X1, X2)).T
        x_del = np.matrix(x_delta(X1, X2)).T
        first = (x_del @ x_del.T)/(x_del.T @ g_del)
        second = (A @ g_del @ g_del.T @ A)/(g_del.T @ A @ g_del)
        A_new = A + first - second
        return A_new
```

In [14]:

```
# as far as each iteration of DFP method is similar we will right it down as a funct
def iteration(X, A, func):
        S = count_s(X[0], X[1], A)
        delta = delta_lambda(X[0], X[1], S[0], S[1])
        a, b = alg_sven(X[0], X[1], delta, LAMBDA0, S)
        _lambda = func(X[0], X[1], S, a, b, EPSILON)
        return X[0]+_lambda*S[0], X[1]+_lambda*S[1]
```

In [15]:

```
# we will use one of two criteria of end, which uses difference of x dots and of f(x
# we won't use the criteria with norm of gradient because from numeric calculating o
def stop_func(X1, X2):
        norma = norm(X2[0]-X1[0], X2[1]-X1[1])/norm(X1[0], X1[1])
        absol = abs(f(X2[0], X2[1])-f(X1[0], X1[1]))
        if norma <= PRECISE and absol <= PRECISE:
                return True
        return False
```

In [16]:

```
# this is main function in program, which minimizes function using DFP method
def dfp():
        X1 = iteration(X0, A0, golden)
        print(f'X1 = {X1}')
        XS = [X0, X1]
        AS = [A0]

        while not stop_func(XS[-2], XS[-1]):
                A_new = count_A(AS[-1], XS[-2], XS[-1])
                X_new = iteration(XS[-1], A_new, golden)

                XS.append(X_new)
                AS.append(A_new)

                print(f'X{len(XS)-1} = {X_new}')

        print(f'\nFunction counted {COUNTER} times')

        dots_x = [X[0] for X in XS]
        dots_y = [X[1] for X in XS]

        plt.figure(0)
        plt.plot(dots_x, dots_y, marker='.')
        plt.show()
```

Обчислення та виведення результатів

В цьому блоці проводимо обчислення та виводимо результати мінімізації функції.

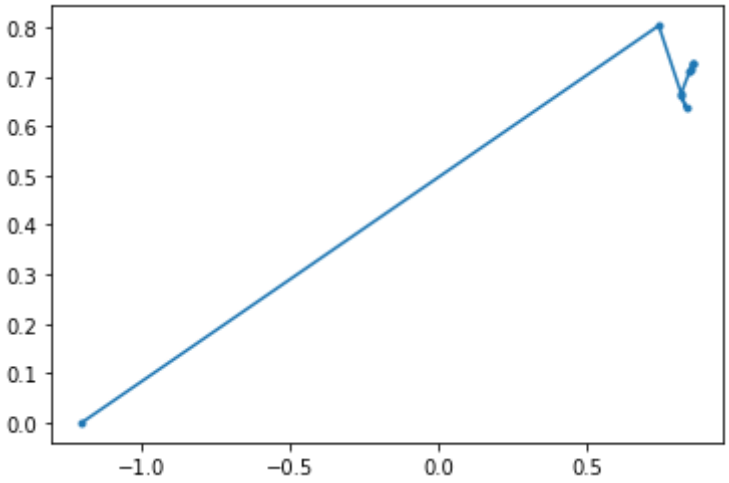
Спершу проведемо мінімізацію функції, коли обидві точки (початкова та глобальний мінімум) знаходяться всередині допустимої області. Це зімітує звичайний алгоритм ДФП, який в теорії має збігтися до точки (1; 1).

In [17]:

```
dfp()
```

```
X1 = (0.7402789318889775, 0.8033166663699736)
X2 = (0.8323682703799736, 0.6357896469299714)
X3 = (0.814405054268473, 0.6627221240520197)
X4 = (0.8154569532167437, 0.6652673821955463)
X5 = (0.8437936087482586, 0.7103920529564688)
X6 = (0.8459521417893647, 0.7154361024710773)
X7 = (0.8526378784893222, 0.7265482921205557)
X8 = (0.8527473280015645, 0.7268354123844356)

Function counted 428 times
```

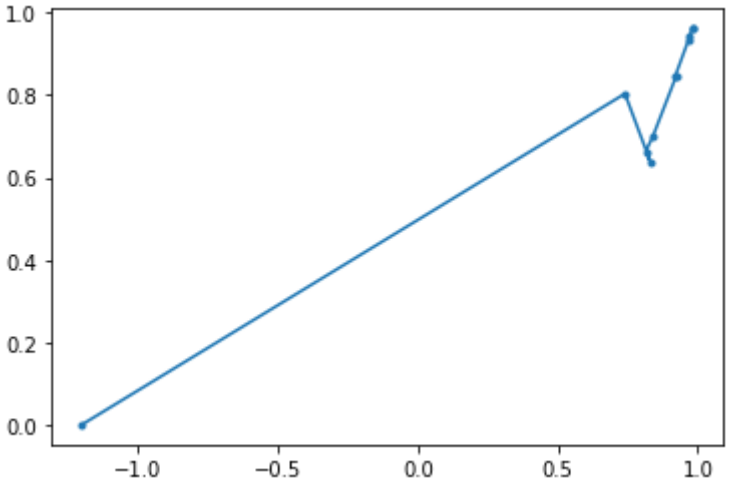


In [18]:

```
COUNTER = 0
R = 0.1
dfp()
```

```
X1 = (0.7402730448254444, 0.8033308854536828)
X2 = (0.8323698648864114, 0.6358066873807248)
X3 = (0.8146387920936329, 0.6631523523529228)
X4 = (0.840947169092329, 0.7011074742014796)
X5 = (0.9237391368582769, 0.8466388996988042)
X6 = (0.920351677469166, 0.8466665989473556)
X7 = (0.9667636031626461, 0.9327356847864468)
X8 = (0.9713577060841225, 0.9437091351468108)
X9 = (0.9801287359554378, 0.9604627258862)
X10 = (0.9805002335753522, 0.9613411849818788)

Function counted 538 times
```

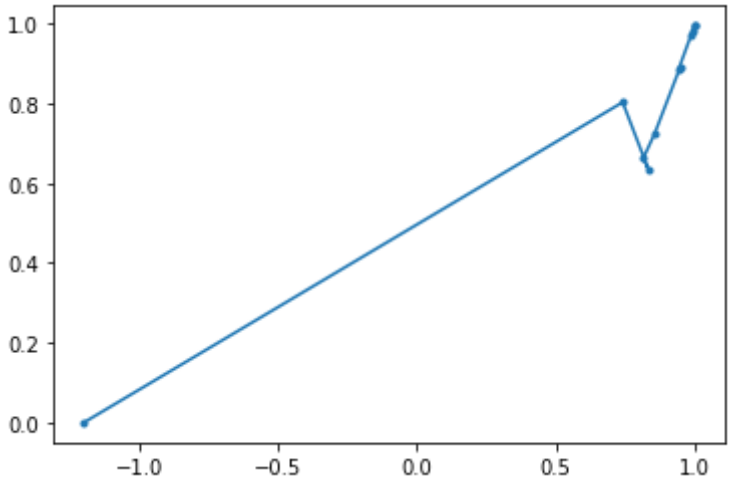


In [19]:

```
COUNTER = 0
R = 0.01
dfp()
```

```
X1 = (0.7402724561018981, 0.8033323073873647)
X2 = (0.8323700244507949, 0.635808391540311)
X3 = (0.8146742392240275, 0.6631769478707875)
X4 = (0.8557940010119564, 0.7252848854862953)
X5 = (0.9472848554496845, 0.891126225209746)
X6 = (0.9414680785374462, 0.8857676800485064)
X7 = (0.9875066197572192, 0.9735968810121471)
X8 = (0.9897386002694334, 0.9796887438450063)
X9 = (0.9976718677591399, 0.9952464269485124)
X10 = (0.9977191416116388, 0.9954398918129838)

Function counted 538 times
```

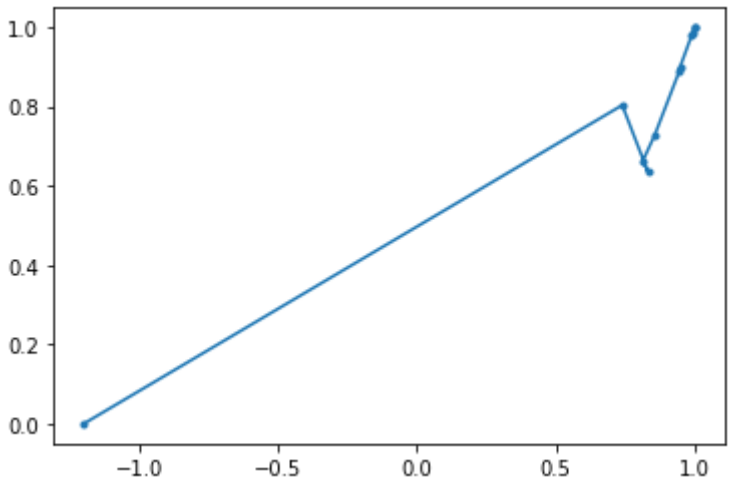


In [20]:

```
COUNTER = 0
R = 0.001
dfp()
```

```
X1 = (0.7402723972293683, 0.8033324495809929)
X2 = (0.8323700404083696, 0.6358085619574223)
X3 = (0.8146777840465508, 0.6631794039386762)
X4 = (0.857544316103455, 0.7282023998965673)
X5 = (0.9505192721752181, 0.897409198723523)
X6 = (0.9445567631606862, 0.8915807941732903)
X7 = (0.9900486161540997, 0.9786976799022932)
X8 = (0.9919807463884752, 0.9841204521528749)
X9 = (0.9995261300691983, 0.998965806452182)
X10 = (0.9995231296806179, 0.9990458831231425)

Function counted 542 times
```

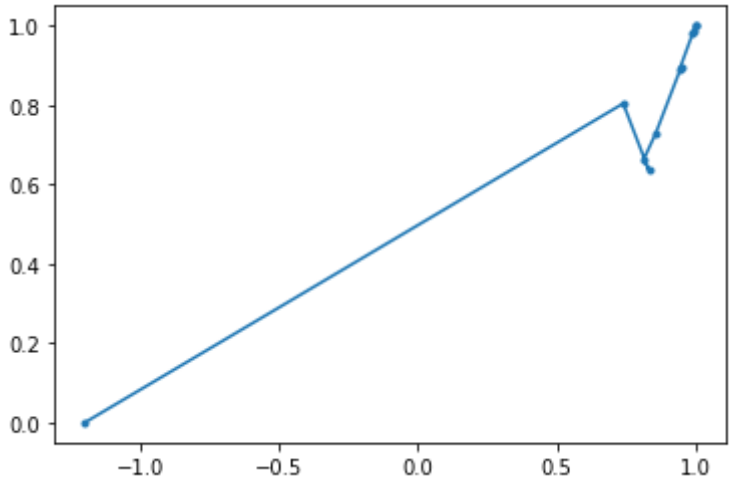


In [21]:

```
COUNTER = 0
R = 0.0001
dfp()
```

```
X1 = (0.7402723913421194, 0.8033324638003458)
X2 = (0.8323700420041408, 0.6358085789991316)
X3 = (0.8146781385298562, 0.6631796495106148)
X4 = (0.8577210059641777, 0.7284977790487716)
X5 = (0.9502484569699472, 0.8968406292996318)
X6 = (0.9442457332407922, 0.8909899996388012)
X7 = (0.9900295031840537, 0.9786416138588216)
X8 = (0.9920068230258599, 0.9841759036849704)
X9 = (0.99968478267378, 0.9992798632297194)
X10 = (0.9996987149658665, 0.9993977280145155)

Function counted 540 times
```



Як бачимо функція дійсно збігається до теоретичних значень з допустимою похибкою, що дає нам можливість стверджувати, що алгоритм працює правильно.

Тепер перебудуємо допустиму область так, що глобальний мінімум знаходився позі нею. В даному випадку це круг з радіусом 1 та центром в точці (-1; 0). В такому випадку алгоритм має збігатися до точки (0; 0).

Також нам треба бути змінити деякі константи для кращих результатів прни виведенні. Дані значення констант підібрані під час туестів програми.

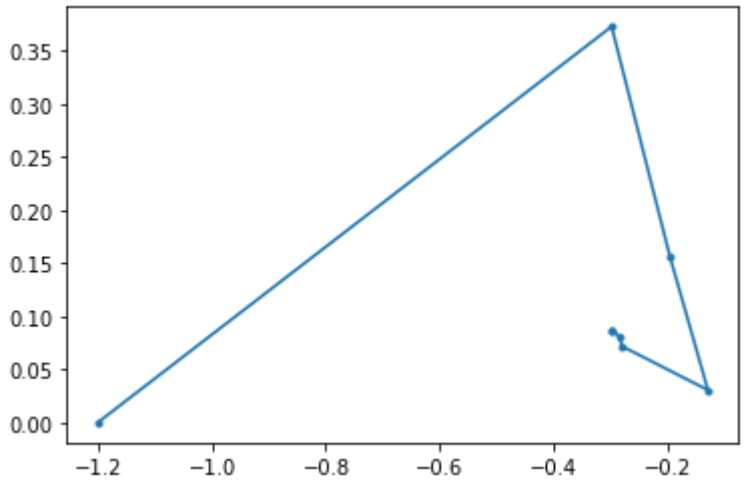
In [22]:
```
SVEN_PAR = 0.25
PRECISE = 0.01
COUNTER = 0
R = 1
```

In [23]:
```
def g(x1, x2):
        return 1-(x1+1)**2-x2**2
```

In [24]:
```
dfp()
```

```
X1 = (-0.2990803436544489, 0.37278515637054843)
X2 = (-0.19706916138212144, 0.15668755137671386)
X3 = (-0.1303002093833711, 0.030689797195361634)
X4 = (-0.27987476079102813, 0.07165076818526475)
X5 = (-0.2865358992782262, 0.08142574061272653)
X6 = (-0.2980089345512121, 0.086807710220959)
X7 = (-0.29839046793009455, 0.08729554747633823)

Function counted 303 times
```

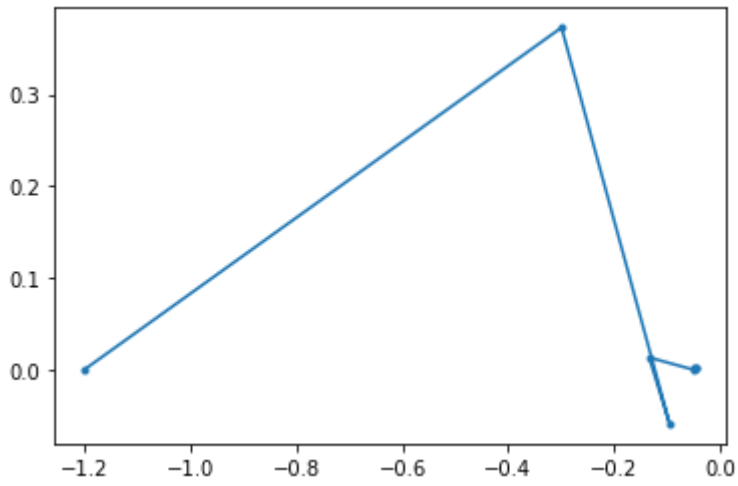


In [25]:

```
COUNTER = 0
R = 0.1
dfp()
```

```
X1 = (-0.2991513513808376, 0.37295671636401556)
X2 = (-0.09500932142700086, -0.0593791194399772)
X3 = (-0.13354717760691867, 0.012914356477282493)
X4 = (-0.04748514369652093, -0.0008389425252339922)
X5 = (-0.04813673168309724, 0.002218745240098822)
X6 = (-0.04662786901539985, 0.0021503958230751317)
X7 = (-0.04662788622287591, 0.0021503981638802015)

Function counted 315 times
```

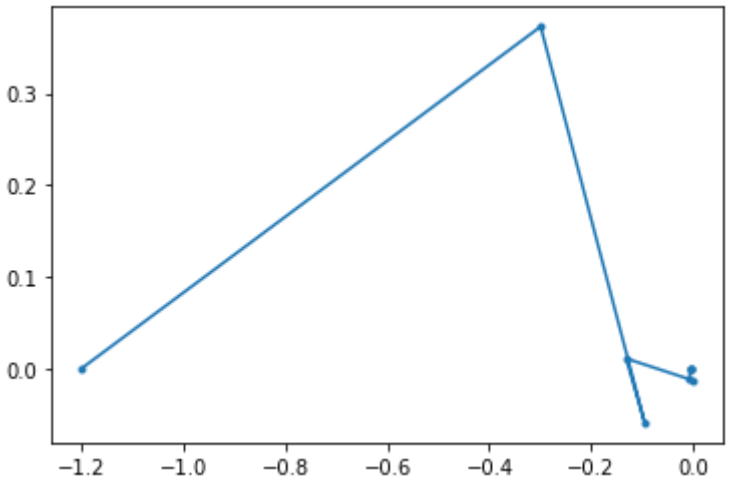


In [26]:

```
COUNTER = 0
R = 0.01
dfp()
```

```
X1 = (-0.29915845754610193, 0.3729738803043474)
X2 = (-0.09500446214834651, -0.05937602922932861)
X3 = (-0.1314110477424555, 0.010703599312458553)
X4 = (-0.0017022830283777157, -0.012940995489728227)
X5 = (-0.006683060489820653, -0.011854775436369527)
X6 = (-0.006108680839691984, 0.00011854791191844537)
X7 = (-0.00496197905670869, 6.132722166003712e-05)
X8 = (-0.0049702089807521724, 5.7736367154724115e-05)

Function counted 300 times
```

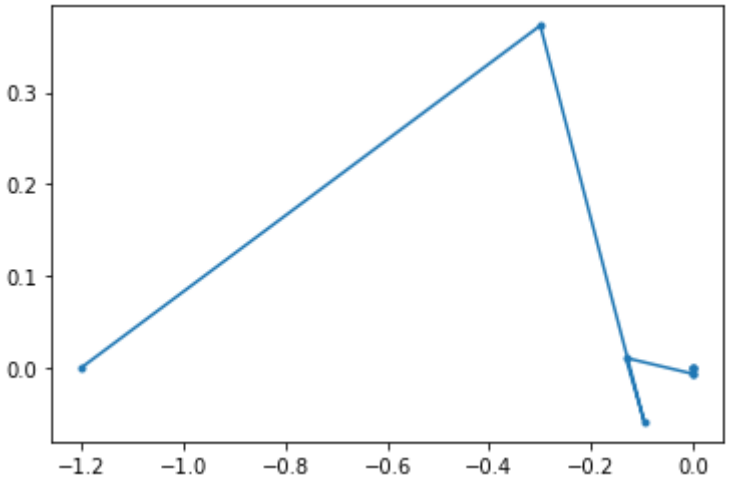


In [27]:

```
COUNTER = 0
R = 0.001
dfp()
```

```
X1 = (-0.2991591682165903, 0.3729755967778239)
X2 = (-0.09500397628374402, -0.059375720239574215)
X3 = (-0.13107258683059148, 0.010244834602268368)
X4 = (-0.0007308728089922556, -0.006960316963539399)
X5 = (-0.0011628123503682876, -0.006891345308344597)
X6 = (-0.0011618681854996 95, 6.9395417555224645e-06)
X7 = (-0.0010825516056117 14, -0.0006486009280631715)
X8 = (-0.0010839526430921685, -0.0006367820648271914)

Function counted 290 times
```

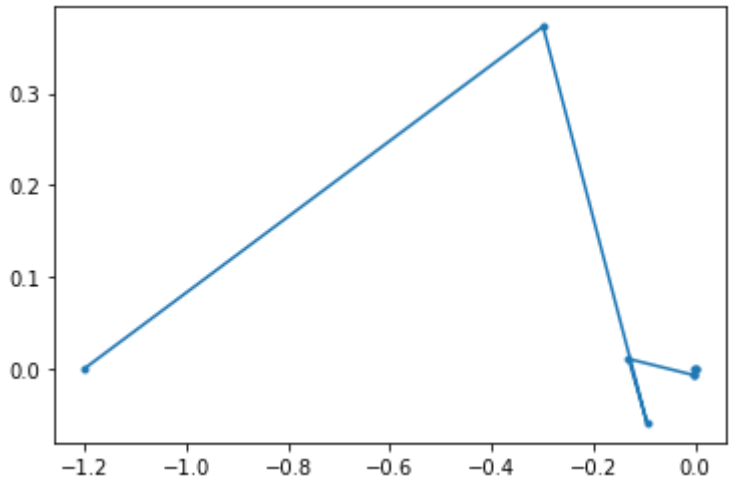


In [28]:

```
COUNTER = 0
R = 0.0001
dfp()
```

```
X1 = (-0.2991592392841782, 0.37297576842596547)
X2 = (-0.09500392769791485, -0.05937568934091114)
X3 = (-0.13106463378739594, 0.010248914645932514)
X4 = (-0.0008361500444913628, -0.007737761068825424)
X5 = (-0.0013159263415845346, -0.0076576084392651345)
X6 = (-0.00128234527584852, 3.57085610699177e-05)
X7 = (-8.978017567263987e-05, 0.0005084378595772065)
X8 = (-9.408670556095369e-05, 0.0005065380867098571)

Function counted 238 times
```



Як бачимо, програма знову збігається до теоретичних значень за оптимальну кількість обчислень значення цільової функції.

Висновки

В даній курсовій роботі нами було розглянуто алгоритми безумовної та умовної оптимізації. Розроблено програму, яка реалізує мінімізацію функції Розенброка методом Девідона-Флетчера-Пауелла із використанням методу внутрішньої точки та методів одномірного пошуку.

Дана програма показує задовільні результати при обчисленні мінімуму функції Розенброка з заданої початкової точки.

Для обчислення оптимальної довжини кроку методом одновимірного пошуку було використано метод золотого перерізу. Також можна використовувати інші методи одновимірного пошуку, такі як дихотомія або ДСК-Пауелл. В даній роботі через занадто малі значення параметрів при обчисленні оптимальної довжини кроку в дейяких випадках ставалася помилка при обчисленні нових точок. Через це для обчислень було обрано метод золотого перерізу.

Під час обчислень було продемонстровано різні варіанти допустимої області, така, коли глобальний мінімум знаходиться всередині області, та така, коли глобальний мінімум лежить поза нею.

Під час тестування програми було обрані оптимальні значення змінних, які відповідають за множник під час обчислення оптимальної довжини кроку алгоритмом Свенна, за крок під час чисельного обчислення похідних, за точність у критеріях закінчення. Таким чином нам вдалося досягти потрібної точності обчислень за 200-400 обчислень цільової функції. Зважаючи на те, що функція Розенброка є достатньо складною для алгоритмів оптимізації через свою яристість можемо стверджувати, що результат задовільний.

Використані джерела

1. Химмельблау Д. Прикладное нелинейное программирование / Химмельблау Д. — М. : Мир, 1975. — 535 с.
2. Методы оптимизации: Конспект лекций / Б.Ю. Лемешко. – Новосибирск: Изд-во НГТУ, 2009. – 126 с.

3. Реклейтис Г. Оптимизация в технике: В 2-х книгах. / Реклейтис Г., Рейвиндран А., Рэгсдел К.; [пер. с англ.]. — М. : Мир, 1986. — 747 с.
4. Матряшин Н. П. Математическое программирование / Н. П. Матряшин, В. К. Макеева. — Харьков : Вища школа, 1978. — 160 с.