УНИВЕРСИТЕТ ИТМО

Студенты: Кочубеев Николай, Михайличенко Глеб, Полухин Максим

Отчет по лабораторной работе №1
Вариант №6

Цель работы

- Реализовать методы одномерной минимизации функции на практике, понять их отличие друг от друга и научиться разумно применять их.

Задачи, решаемые при выполнении работы

1. Реализовать алгоритмы одномерной минимизации функции без производной:
   - Методом Дихотомии;
   - Методом золотого сечения;
   - Методом Фибоначчи;
   - Методом парабол;
   - Комбинированным методом Брента;
2. Сравнить методы по количеству итераций и количеству вычислений функций;
3. Протестировать реализованные алгоритмы.

Описание задачи

Весьма неприятно оказываться в ситуации, где ты со всех сторон окружен радиацией. Наверное не хотелось бы оставаться в ней долго. Именно так и подумал обычный инженер с дозиметром в руках после аварии на АЭС, где он работал до этого несчастного случая. Чтобы понять куда ему дальше двигаться и поскорее убраться от источника излучения, он решил сделать замеры направленного эквивалента дозы вокруг себя. Инженер он был опытный, и в его голове сразу возникла модельная функция этой величины от угла его первоначального направления движения:

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

Ему нужно быстро сообразить в каком направлении двигаться наиболее безопасно.

Теоретические значения:

a = -3
b = -1
x = -2.289
y(x) = -3.945

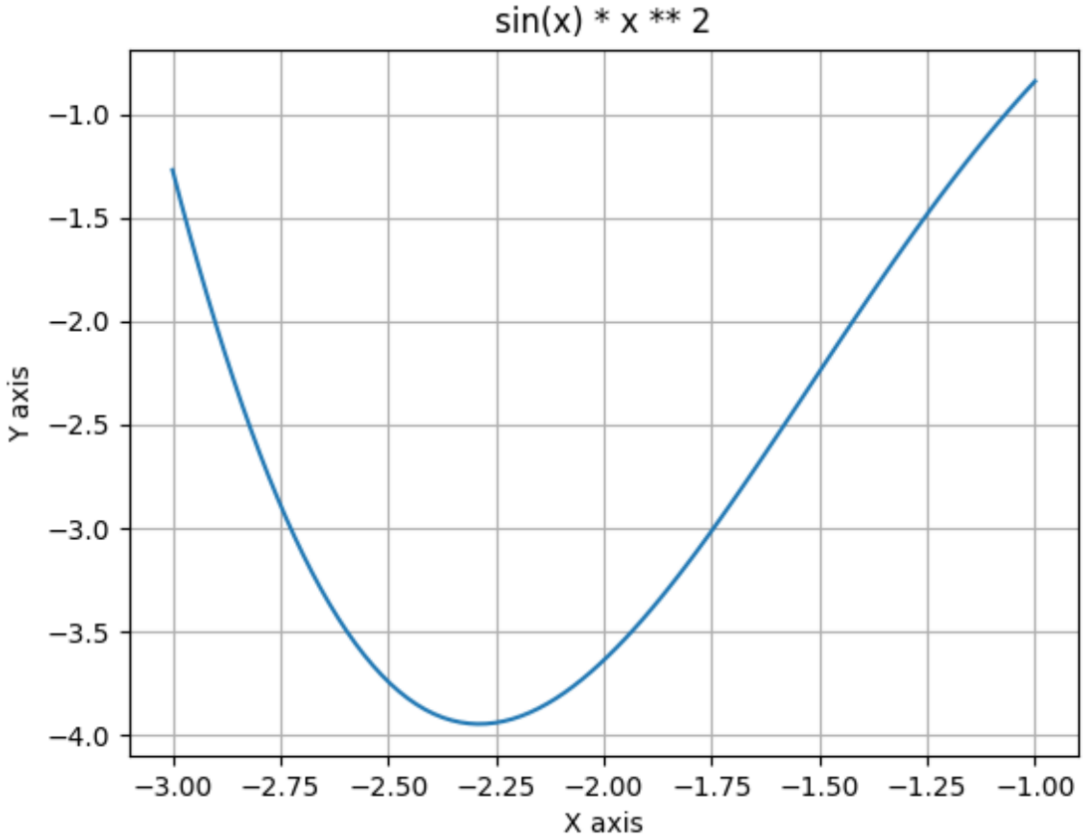
Описание задачи на Python с построением графика функции

```
class Optimization:
   def __init__(self, function, a, b, epsilon):
       self.function = function
       self.a = a
       self.b = b
       self.epsilon = epsilon
       self.epsilons = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]
       self.ratio = 0.61803398874
       self.u = lambda x1, x2, x3, f1, f2, f3: x2 - ((x2 - x1) ** 2 *
(f2 - f3) - (x2 - x3) ** 2 * (f2 - f1)) / (
               2 * ((x2 - x1) * (f2 - f3) - (x2 - x3) * (f2 - f1))) \
           if (2 * ((x2 - x1) * (f2 - f3) - (x2 - x3) * (f2 - f1))) != 0
\
           else None
```

```
def plot_function(self):
   vectorize_function = np.vectorize(self.function)
   array = np.linspace(self.a, self.b, 100)
   plt.title("sin(x) * x ** 2")
   plt.xlabel("X axis")
   plt.ylabel("Y axis")
   plt.grid()
   plt.plot(array, vectorize_function(array))
   plt.show()
```

```
optimization = Optimization(lambda x: sin(x) * x ** 2, -3, -1, 1e-5)
optimization.plot_function()
```

График функции на заданном отрезке [-3; -1]:



sin(x) * x ** 2
Y axis
−1.0
−1.5
−2.0
−2.5
−3.0
−3.5
−4.0
−3.00
−2.75
−2.50
−2.25
−2.00
−1.75
−1.50
−1.25
−1.00
X axis

1. Метод Дихотомии

Метод Дихотомии заключается в определении корня уравнения на отрезке при помощи двух точек, отстоящих на величину δ < 2ε в обе стороны от середины интервала неопределенности. На каждой итерации рассчитываются значения функций в обеих точках, после чего одна из точек (или обе, если значения функций совпали) сдвигается.
Алгоритм на языке Python:

```python
def calculate_dichotomy(self):
    print('dichotomy method')
    a = self.a
    b = self.b
    prev_length = b - a
    iterations = 0
    while (b - a) / 2 > self.epsilon:
        x1 = (a + b) / 2 - self.epsilon / 3
        x2 = (a + b) / 2 + self.epsilon / 3
        if self.function(a) < self.function(b):
            b = x2
        elif self.function(a) > self.function(b):
            a = x1
        else:
            a = x1
            b = x2
        iterations += 1
        print(iterations, ': ', b - a, ' ', (b - a) / prev_length)
        prev_length = b - a
    x = (a + b) / 2
    print('x = ', x)
    print('f(x) = ', self.function(x))
    print('iterations = ', iterations, '\n')
    return iterations
```

x = -2.2889280224990842
f(x) = -3.9453016252720152
iterations = 18

Длина отрезка и её отношение к предыдущей длине на каждой итерации:

```
dichotomy method
1 :  1.0000033333333334   0.5000016666666667
2 :  0.5000049999999998   0.500003333322222
3 :  0.25000583333333326   0.5000066666000007
4 :  0.12500624999999976   0.5000133330222287
5 :  0.06250645833333346   0.5000266653334019
6 :  0.03125656250000031   0.5000533278227955
7 :  0.01563161458333351   0.5001066442713704
8 :  0.00781914062500011   0.5002132430604391
9 :  0.00391290364583341   0.5004263043080076
10 :  0.00195978515625006   0.5008518822938317
11 :  0.0009832259114581632   0.5017008667110793
12 :  0.0004949462890624368   0.503390200862802 4
13 :  0.0002508064778643515   0.5067347374993908
14 :  0.0001287365722655309 2   0.5132904594878845
15 :  6.770161946612063e-05   0.5258926680638961
16 :  3.718414306641549e-05   0.5492356513720213
17 :  2.192540486678496e-05   0.5896439465506431
18 :  1.429603576674765e-05   0.6520306399634551
```

2. Метод Золотого Сечения

$$\frac{b-a}{b-x_1} = \frac{b-a}{x_2-a} = \Phi = \frac{1+\sqrt{5}}{2} = 1.618\ldots$$

$$\begin{aligned} x_1 &= b - \tfrac{(b-a)}{\Phi} \\ x_2 &= a + \tfrac{(b-a)}{\Phi} \end{aligned}$$

На первой итерации заданный отрезок делится двумя симметричными относительно его центра точками и рассчитываются значения в этих точках. После чего тот из концов отрезка, к которому среди двух вновь поставленных точек ближе оказалась та, значение в которой минимально, сохраняют, другой отбрасывают. На следующей итерации ищем всего одну новую точку. Работа метода завершается по достижении заданной точности

Алгоритм на языке Python:

```python
def calculate_golden_ratio(self):
   print('golden ratio method')
   a = self.a
   b = self.b
   prev_length = b - a
   iterations = 0
   x1 = b - (b - a) * self.ratio
   x2 = a + (b - a) * self.ratio
   f1 = self.function(x1)
   f2 = self.function(x2)
   while (x2 - x1) / 2 > self.epsilon:
       if f1 < f2:
           b = x2
           x2 = x1
           x1 = b - (b - a) * self.ratio
           f2 = f1
           f1 = self.function(x1)
       else:
           a = x1
           x1 = x2
           x2 = a + (b - a) * self.ratio
           f1 = f2
           f2 = self.function(x2)
       iterations += 1
       print(iterations, ': ', b - a, ' ', (b - a) / prev_length)
       prev_length = b - a
   x = (a + b) / 2
   print('x = ', x)
   print('f(x) = ', self.function(x))
   print('iterations = ', iterations, '\n')
   return iterations
```

x = -2.288945230448066
f(x) = -3.9453016242673313
iterations = 21

Длина отрезка и её отношение к предыдущей длине на каждой итерации:

```
golden ratio method
1 :  1.23606797748   0.61803398874
2 :  0.7639320224757491   0.6180339887400002
3 :  0.47213595497690264   0.6180339887400002
4 :  0.2917960674819442   0.61803398874
5 :  0.18033988748451213   0.6180339887399998
6 :  0.11145618002079027   0.6180339889053236
7 :  0.06888370750797268   0.6180339887400015
8 :  0.04257247251035201   0.6180339887400025
9 :  0.026311234980718634   0.6180339883787755
10 :  0.016261237503809145   0.6180339887400073
11 :  0.010049997476327732   0.6180339887400053
12 :  0.0062112400271217005   0.6180339887399938
13 :  0.003838757448983543   0.6180339887399956
14 :  0.00237248257800049   0.6180339887399489
15 :  0.001466274870897788   0.6180339887399945
16 :  0.0009062077171564376   0.6180339956324656
17 :  0.0005600671700611848   0.6180339887400242
18 :  0.000346140553308949   0.6180339998702918
19 :  0.0002139266268263995   0.6180339887405754
20 :  0.00013221393032303297   0.6180340067266333
21 :  8.171270272461229e-05   0.6180339887405731
```

3. Метод Фибоначчи

В силу того, что в бесконечности отношение соседних чисел Фибоначчи стремится к золотому сечению, одноименный метод может быть трансформирован в так называемый метод Фибоначчи. Однако при этом в силу свойств чисел Фибоначчи количество итерации строго ограничено.
Алгоритм на языке Python:

```
def calculate_fibonacci(self, n):
    print('fibonacci method')
    a = self.a
    b = self.b
    prev_length = b - a
    sequence = self.get_fibonacci_sequence(n)
    x1 = a + (b - a) * (sequence[n - 1] / sequence[n + 1])
    x2 = a + (b - a) * (sequence[n] / sequence[n + 1])
    f1 = self.function(x1)
    f2 = self.function(x2)
    iterations = 0
    while n > 0:
        n -= 1
        if f1 < f2:
            b = x2
            x2 = x1
            x1 = a + (b - x2)
            f2 = f1
            f1 = self.function(x1)
        else:
            a = x1
            x1 = x2
            x2 = b - (x1 - a)
            f1 = f2
            f2 = self.function(x2)
        iterations += 1
        print(iterations, ': ', b - a, ' ', (b - a) / prev_length)
        prev_length = b - a
    x = (a + b) / 2
    print('x = ', x)
    print('f(x) = ', self.function(x))
    print('iterations = ', iterations, '\n')
    return iterations
```

x = -2.2889293451987593
f(x) = -3.9453016252837054
iterations = 27
Длина отрезка и её отношение к предыдущей длине на каждой итерации:

```
fibonacci method
1 :  1.2360679775086452   0.6180339887543226
2 :  0.7639320224913553   0.6180339887383033
3 :  0.4721359550172899   0.6180339887802421
4 :  0.29179606747406517   0.6180339886704443
5 :  0.18033988754322472   0.618033988957899
6 :  0.11145617993084045   0.6180339882053332
7 :  0.06888370761238427   0.6180339901755759
8 :  0.04257247231845618   0.6180339850174132
9 :  0.026311235293928092   0.6180339985216585
10 :  0.016261237024528086   0.6180339631670859
11 :  0.010049998269400007   0.6180340557265609
12 :  0.006211238755128079   0.6180338134027256
13 :  0.0038387595142719277   0.618034447814874
14 :  0.0023724792408561513   0.6180327869030691
15 :  0.0014662802734157765   0.6180371352318528
16 :  0.0009061989674403748   0.6180257511951224
17 :  0.0005600813059754017   0.6180555552357253
18 :  0.000346117661464973   0.6179775289271559
19 :  0.0002139636445104287   0.6181818159894211
20 :  0.00013215401695454432   0.6176470645605546
21 :  8.180962755588439e-05   0.6190476040090679
22 :  5.034438939865993e-05   0.6153846546272262
23 :  3.146523815722446e-05   0.6249998963749872
24 :  1.8879151241435466e-05   0.6000002652800766
25 :  1.2586086915788997e-05   0.6666659297778909
26 :  6.293064325646469e-06   0.5000016580015783
27 :  6.293022590142527e-06   0.9999933680156785
```

4. Метод Парабол

Поскольку в данной ситуации функция унимодальна, мы можем аппроксимировать ее при помощи квадратичной функции, так, чтобы вершина параболы и искомая точка экстремума совпали. Если известно, что парабола проходит через три различные точки x1, x2, x3, то ее вершину u можно вычислить. Если $x1 < x2 < x3$, а также $f2 < f1$ и $f2 < f3$, тогда вершина параболы u гарантированно попадает в интервал $(x1, x3)$. В рамках метода парабол целевая функция приближается параболой, соответственно, в качестве приближения к точке минимума выбирается вершина параболы.

Алгоритм на языке Python:

```python
def calculate_parabola(self):
   print('parabola method')
   x1 = self.a
   x3 = self.b
   x2 = (x3 + x1) / 2
   x_i = 0
   prev_length = x3 - x1
   iterations = 0
   f1 = self.function(x1)
   f2 = self.function(x2)
   f3 = self.function(x3)
   u = self.u(x1, x2, x3, f1, f2, f3)
   fu = self.function(u)
   while abs(x_i - u) >= self.epsilon:
       x_i = u
       iterations += 1
       if fu <= f2:
           if u >= x2:
               x1 = x2
               f1 = f2
           else:
               x3 = x2
               f3 = f2
           x2 = u
           f2 = fu
       else:
           if x2 < u:
               x3 = u
               f3 = fu
           else:
               x1 = u
               f1 = fu
       print(iterations, ':  ', x3 - x1, ' ', (x3 - x1) / prev_length)
       prev_length = x3 - x1
       u = self.u(x1, x2, x3, f1, f2, f3)
       fu = self.function(u)
```

Длина отрезка и её отношение к предыдущей длине на каждой итерации:

```
1 :   1.0   0.5
2 :   0.9584908641085947   0.9584908641085947
3 :   0.7682177231322398   0.8014867453605719
4 :   0.7445388062904468   0.96917681520644
5 :   0.7202872200630677   0.9674273711155379
6 :   0.7153961177488282   0.9932095111810935
7 :   0.7124296987141672   0.9958534594177061
8 :   0.711640219720719   0.9988918499679715
9 :   0.7112634784396801   0.9994706014772651
10 :   0.711146675524954   0.9998357810877873
11 :   0.7110972604509165   0.9999305135273239
```

5. Метод Брента

Метод Брента эффективно комбинирует две стратегии: надежность и гарантию сходимости метода золотого сечения, и суперлинейную скорость сходимости метода парабол. Эффективное использование данных методов на различных итерациях позволяет нивелировать их недостатки(неустойчивость метода парабол на начальных этапах, линейную скорость сходимости метода золотого сечения

```
def calculate_brent(self):
   print('brent combined method')
   a = self.a
   b = self.b
   iterations = 0
   x = w = v = a + self.ratio * (b - a)
   d = e = b - a
   prev_length = 1
   fx = self.function(x)
   fw = self.function(x)
   fv = self.function(x)
   while max(abs(x - a), abs(b - x)) >= self.epsilon:
       g = e / 2
       e = d
       u = self.u(x, w, v, fx, fw, fv)
       if u is None:
           if x >= (a + b) / 2:
               u = x - self.ratio * (x - a)
               e = x - a
           else:
               u = x + self.ratio * (b - x)
               e = b - x

       d = abs(u - x)
       fu = self.function(u)
       if fu > fx:
           if u >= x:
               b = u
           else:
               a = u
           if fu <= fw or w == x:
               fv = fw
               v = w
               w = u
               fw = fu
           else:
               if fu <= fv or x == v or v == w:
                   v = u
                   fv = fu
       else:
           if u >= x:
               a = x
           else:
```

```
            b = x

        fv = fw
        fw = fx
        fx = fu
        v = w
        w = x
        x = u
        iterations += 1

        print(iterations, ':  ', b - a, ' ', (b - a) / prev_length)
        prev_length = b - a

    print('x = ', x)
    print('f(x) = ', self.function(x))
    print('iterations = ', iterations, '\n')
    return iterations
```

x = -2.288929728073737
f(x) = -3.9453016252843254
iterations = 9
Длина отрезка и её отношение к предыдущей длине на каждой итерации:

```
brent combined method
1 :   1.23606797748   1.23606797748
2 :   0.7639320224757491   0.6180339887400002
3 :   0.47213595497690264   0.6180339887400002
4 :   0.25524550582765615   0.5406186568446011
5 :   0.2508573332152384   0.9828080318272844
6 :   0.013613346749189237   0.05426728640820252
7 :   0.011954854570733975   0.8781716054831237
8 :   3.623723983947613e-05   0.0030311736228215216
9 :   4.4088546444776e-06   0.12166640351218695
```

Сравнение методов

Таблица 1: Зависимость x от ε для каждого метода

epsilon	dichotomy	g_ratio	fibonacci	parabola	brent
1e-1	-2.3020833333	-2.1458980337	-2.2889327304	-2.2554611937	-2.2889615663
1e-2	-2.2919921875	-2.2705098312	-2.2889327304	-2.2846038822	-2.2889297379
1e-3	-2.2884780273	-2.2879573100	-2.2889327304	-2.2883597802	-2.2889297379
1e-4	-2.2888697652	-2.2889704810	-2.2889327304	-2.2889027395	-2.2889297379
1e-5	-2.2889280224	-2.2889452304	-2.2889327304	-2.2889259906	-2.2889297280

Таблица 2: Зависимость кол-ва итераций от ε для каждого метода

epsilon	dichotomy	g_ratio	fibonacci	parabola	brent
1e-1	4	2	27	2	6
1e-2	8	7	27	4	8
1e-3	11	12	27	6	8
1e-4	14	17	27	9	8
1e-5	18	21	27	11	9

Вывод:

Использование всех пяти методов позволило нам верно определить минимум функции эквивалента дозы радиации на заданном интервале, что дает нам право говорить о верности реализации представленных для исследования методов оптимизации. Быстрее и точнее всего справится с поставленной задачей нахождения безопасного направления выхода из радиоактивной зоны позволил метод Брента, которому понадобилось на это 9 итераций.