Содержание

Задание

Для унимодальной на отрезке $[e; f]$ функции $f(x) = ax^3 + bx^2 + cx + d$ найти минимум с использованием следующих методов одномерной оптимизации. Использовать методы:

- **Метод 1:** Метод равномерного поиска
- **Метод 2:** Метод ломаных
- **Метод 3:** Метод секущих
- **Метод 4:** Метод Ньютона

Построить математическую модель задачи, решить ее аналитически. Разработать программу одномерной минимизации на Python с использованием библиотек Numpy, Scipy, Matplotlib и убедиться в правильности результата, сравнив его с аналитическим решением.

Алгоритм

1. Выбрать самостоятельно и описать исходную функцию, привести примеры задач, в которых могут возникнуть подобные исходные данные. Привести примеры задач, которые решаются с использованием одномерной оптимизации. Описать теоретически используемые методы решения задачи.
2. Проанализировать исходные данные. Проверить уникальность функции. Изобразить исходную функцию при помощи библиотеки matplotlib, привести код. Сделать предположения о минимуме.
3. Решить задачу аналитически. Описать нахождение минимума каждым методом вручную. Изобразить результаты графически при помощи библиотеки matplotlib, привести код для построения значений. (Аналитическое решение можно выполнить от руки, приложить в отчет отсканированные фото)
4. Решить задачу на языке python с использованием библиотеки питона, привести код, привести результат решения. Полученный результат изобразить графически при помощи библиотеки matplotlib, привести код для построения значений.

5. Решить задачу на языке python с использованием библиотеки scipy и matplotlib. Полученный результат изобразить графически при помощи библиотеки matplotlib, привести код для построения значений.

6. Сравнить решения, полученные вручную и с помощью решения на python, построить решения графически. Оценить точность.

7. Сделать вывод.

Ход выполнения работы

Теоретическая справка

Исходная функция.

Дана унимодальная кубическая функция $f(x) = ax^3 + bx^2 + cx + d$ на отрезке $[e; f]$. Унимодальность означает, что на этом отрезке функция имеет только один экстремум (минимум или максимум), что делает её пригодной для применения методов одномерной оптимизации.

Примеры задач, где возникают подобные исходные данные.

- **Экономика и управление**
  Оптимизация объёма производства, если затраты описываются кубической функцией (например, из-за нелинейных эффектов масштаба). На отрезке $[e; f]$ требуется найти объём, минимизирующий себестоимость.

- **Физика и инженерия**
  Расчёт момента времени, когда скорость движения объекта с кубической зависимостью координаты от времени достигает экстремума на заданном интервале $[e; f]$.

- **Биологическое моделирование**
  Поиск оптимальной концентрации вещества, при которой скорость химической реакции (описываемая кубической функцией) максимальна в диапазоне $[e; f]$.

- **Материаловедение**
  Определение параметра (например, толщины материала), при котором напряжение в конструкции, зависящее от параметра кубически, достигает минимального значения на интервале $[e; f]$.

Примеры задач, решаемых одномерной оптимизацией.

- **Минимизация функции затрат**
  Найти точку $x \in [e, f]$, где функция затрат $f(x)$ достигает минимума.
  Методы: золотое сечение, параболы.

- **Максимизация прибыли**
  Определить цену товара x, максимизирующую прибыль $f(x)$, если зависимость прибыли от цены задана аналитически.

- **Инженерный дизайн**
  Оптимизация длины рычага в механизме для минимизации требуемого усилия, где усилие выражается функцией $f(x)$.

- **Финансовый анализ**
  Поиск оптимальной доли инвестиций в актив, если доходность зависит от этой доли кубически на интервале допустимых значений.

- **Научные эксперименты**
  Калибровка параметра эксперимента (например, температуры) для минимизации погрешности измерений, заданной функцией $f(x)$.

Примечание.

Унимодальность кубической функции на $[e; f]$ может быть обеспечена, даже если её производная (квадратичная функция) имеет два корня, но только один из них попадает в заданный отрезок. Это позволяет гарантировать корректность применения методов одномерной оптимизации.

Утилиты

Для начала заведём вспомогательный класс для хранения данных в виде точек:

```
# Define a helper class for points
# of type (x, y)
@dataclass
class Point:
    x: float
    y: float

    def __str__(self):
        return f'(x={self.x}; y={self.y})'

    def __lt__(self, other):
        return self.y <= other.y
```

```
    def __format__(self, format_spec):
        format_x = format(self.x, format_spec)
        format_y = format(self.y, format_spec)
        return  f'(x={format_x}; y={format_y})'
```

А также абстрактный класс со всеми указаными методами:

```
# Define an abstract class
class Optimization(ABC):
    def __init__(self, func) -> None:
        self.func = func

    # Методравномерногопоиска
    @abstractmethod
    def uniform_search(self, a: float,
                             b: float,
                             n: int,
                             add_to_comparison: bool,
                             *args):
        pass

    # Методломаных
    @abstractmethod
    def broken_lines(self):
        pass

    # Методсекущих
    @abstractmethod
    def secant(self):
        pass

    # МетодНьютона
    @abstractmethod
    def newton(self):
        pass
```

И последний вспомогательный класс для сравнения полученных данных в дальнейшем:

```
class Comparison:
    @dataclass
    class Method:
        name: str
        implementation_type: str
        result: Point
        execution_time: float
        details: str = ''

    def __init__(self) -> None:
        self.data = []
```

```
    def add_data(self, method: Method) -> None:
        self.data.append(method)

    def get_data(self) -> list[Method]:
        return self.data
```

Исходные данные

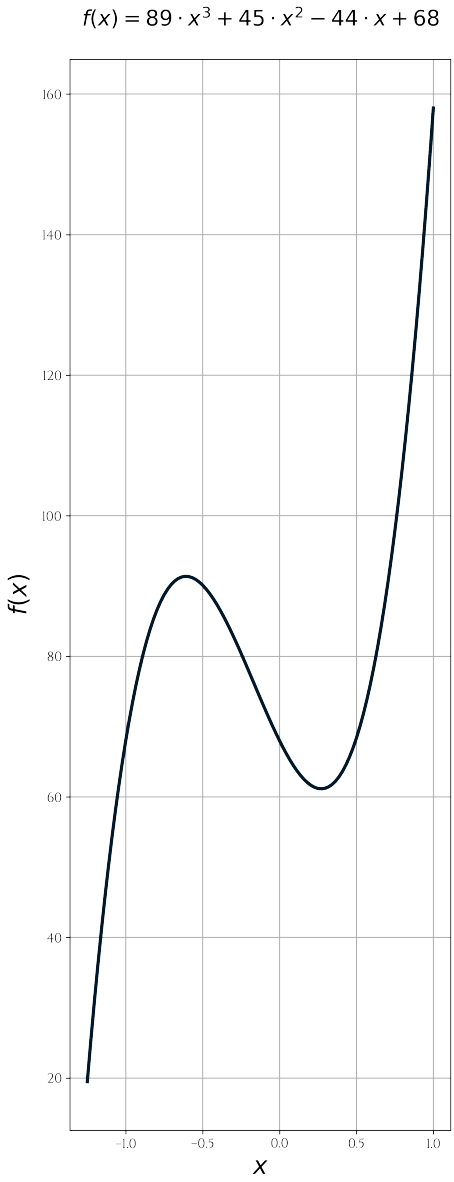
Случайно выберем коэффициенты a, b, c, d и запишем их в коде:

```
# define a function
def func(x, a, b, c, d) -> float:
    return a * x**3 + b * x**2 + c * x + d

# randomly choose coefficients
A, B, C, D = 89, 45, -44, 68

INTERVAL = (-0.5, 1)

# create a Comparison object
cmpr = Comparison()
```

Анализ данных

Построим график заданной функции с выбранными коэффициентами:

```
myfunc = lambda x : func(x, A, B, C, D)

# create plot
with plt.rc_context(classic_style): # use context for styles not to interfere
    fig, ax = plt.subplots(figsize=(6, 15))

    x_values = np.linspace(-1-0.25, 0.75+0.25, 100)
    y_values = myfunc(x_values)

    ax.plot(x_values, y_values, color=RICH_BLACK, linewidth=3)

    ax.set_title(f'$ f(x) = {A} \\cdot x^3 {'+' if B > 0 else ''} {B} \\cdot x^2 {'+
    decorate_regular_plot(ax, '$x$', '$f(x)$')

    ax.grid(linewidth=1)

    if SAVE:
        plt.savefig(f'{IMAGES_PATH}/initial_f_plot.png',
                    dpi=300, transparent=True)

    plt.show()
```



Как видно из графика, выбранный промежуток $([-0.5, 1])$ удовлетворяет условию унимодальности.

«На глаз» можно сделать предположение, что минимум находится где то в районе точки $x = 0.3$.

Метод равномерного поиска

Теория

Метод равномерно поиска (метод перебора) - простейший из методов поиска значений действительно-значных функций по какому-либо из критериев сравнения (на максимум, на минимум, на определённую константу). Применительно к экстремальным задачам является примером прямого метода условной одномерной пассивной оптимизации.

Проиллюстрируем суть метода равномерного поиска посредством рассмотрения задачи нахождения минимума.
Пусть задана функция $f(x) : [a, b] \to \mathbb{R}$ и задача оптимизации выглядит так: $f(x) \to \min$. Пусть также задано число наблюдений n.

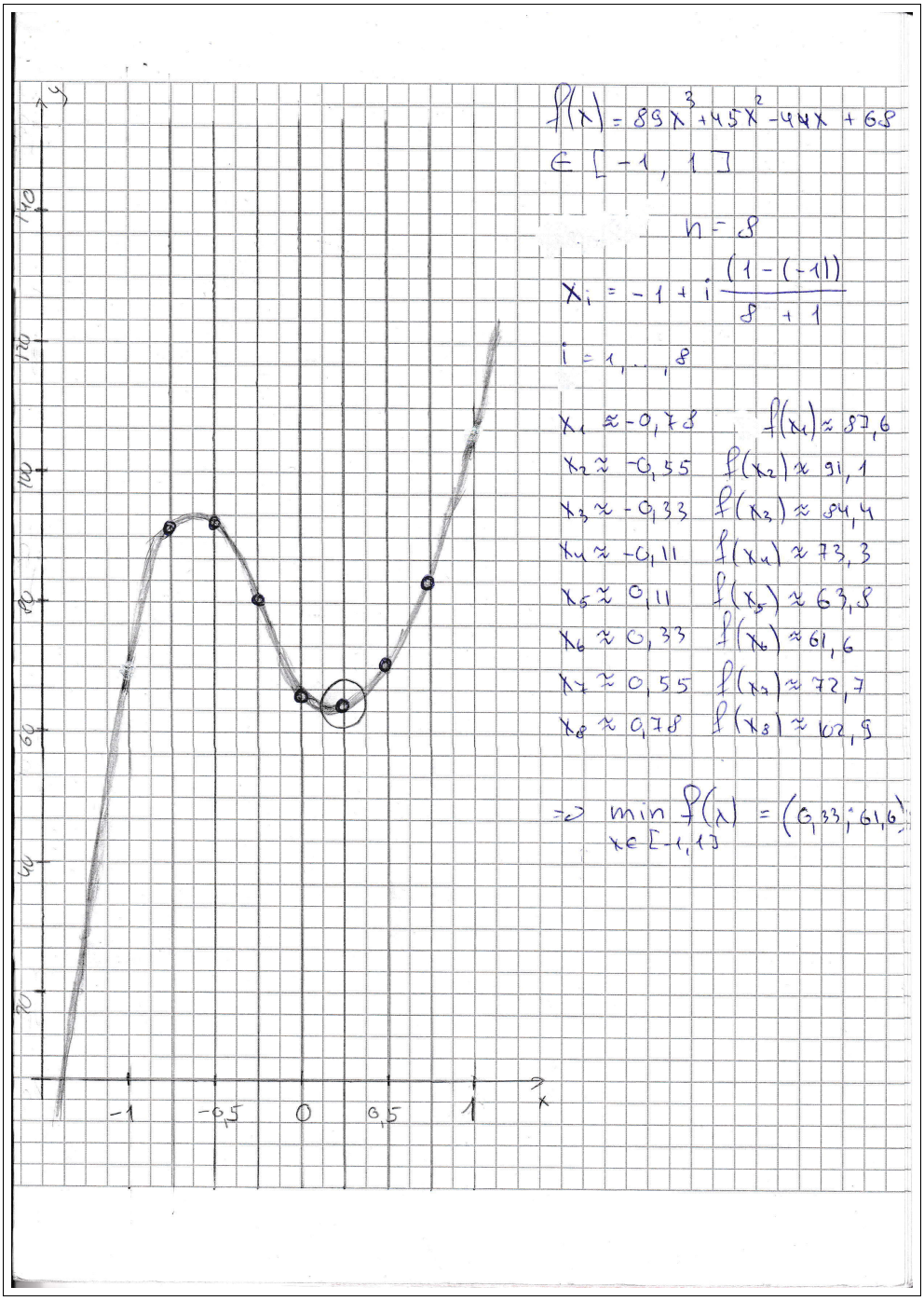
Тогда отрезок $[a, b]$ разбивают на $(n + 1)$ равных частей точками деления:

$$x_i = a + i\frac{(b-a)}{(n+1)}, \quad i = 1, \ldots, n$$

Вычисляя значения $F(x)$ в точках x_i, $i = 1, \ldots, n$, найдем путем сравнения точку x_m, где m — это число от 1 до n такое, что

$$F(x_m) = \min F(x_i) \text{ для всех } i \text{ от 1 до } n.$$

Аналитическое решение



Реализация метода

Теперь наследуем наш ранее объявленный абстрактный класс и реализуем описанный метод:

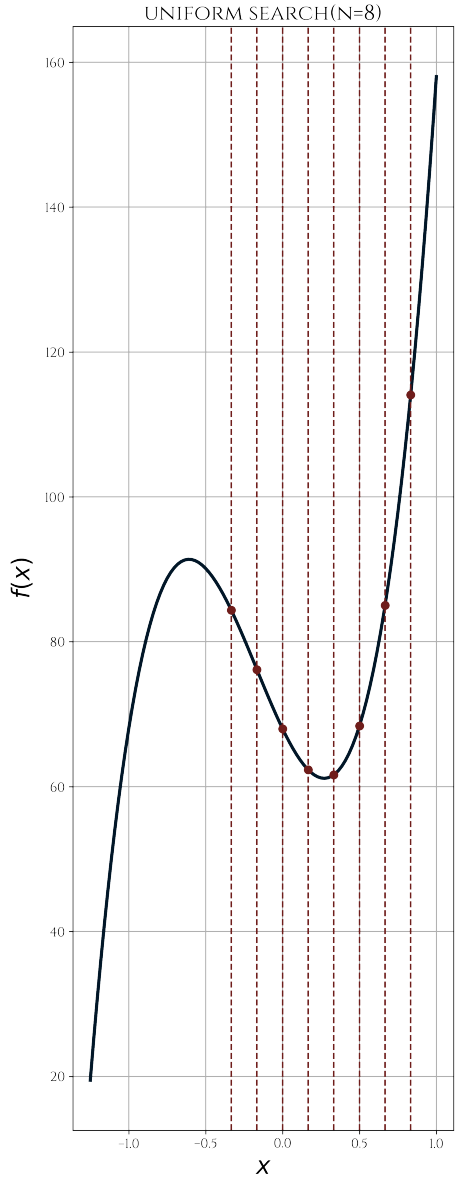
```
# class for analytical implementations
class OptimizationAnalytical(Optimization):
    # Методравномерногопоиска
    def uniform_search(self, a: float,
                             b: float,
                             n: int = 1000,
                             add_to_comparison: bool = True,
                             showplot: bool = False):

        # start timer
        start_time = time.perf_counter()

        points, minpoint = [], None
        for i in range(1, n + 1):
            x = a + i * (b - a)/(n + 1)
            y = self.func(x)
            point = Point(x, y)
            points.append(point)

            if minpoint is None:
                minpoint = point
            else:
                minpoint = min(minpoint, point)

        # end timer
        result_time = time.perf_counter() - start_time

        # add data to Comparison class
        if add_to_comparison:
            method = Comparison.Method('uniform_search', 'analytical',
                                       minpoint, result_time)
            cmpr.add_data(method)

        if showplot:
            # create plot
            with plt.rc_context(classic_style): # use context for styles not to inte
                _, ax = plt.subplots(figsize=(6, 15))

                x_values = np.linspace(-1-0.25, 0.75+0.25, 100)
                y_values = myfunc(x_values)

                # plot main function
                ax.plot(x_values, y_values, color=RICH_BLACK, linewidth=3)

                # plot vertical lines with dots
                for i, point in enumerate(points):
                    # plot lines
                    ax.axvline(x=point.x, color=RED, linestyle='--')
```

```
                    # plot dots
                    ax.scatter(point.x, point.y, color=RED,
                               s=50, zorder=n + 1 + i + 1)

                ax.set_title(f'uniform search(n={n})')
                decorate_regular_plot(ax, '$x$', '$f(x)$')

                ax.grid(linewidth=1)

                if SAVE:
                    plt.savefig(f'{IMAGES_PATH}/uniform_search_f_plot(n={n}).png',
                                dpi=300, transparent=True)

                plt.show()

        return [minpoint, result_time]
```

И запустим на данных, разобранных вручную:

```
# create optimizer and find minimum
optimizer = OptimizationAnalytical(func=myfunc)

min_point, elapsed_time = optimizer.uniform_search(*INTERVAL, showplot=True,
                                                   n=8, add_to_comparison=False)
print(f'Minimum found at: {min_point:.6f}')
```

↓



В результате получим график и искомый минимум:

Minimum found at: (x=0.333333; y=61.629630)

Что не сильно отличается от минимумов, найденных «на глаз» и вручную.

Теперь же запустим данный метод на $n = 1000$ (в дальнейшем для всех методов будет использоваться это значение, если не указано иное) и засечем время его исполнения:

```
min_point, elapsed_time = optimizer.uniform_search(*INTERVAL) # n = 1000 (default)
print(f'Minimum found at: {min_point:.6f} (execution time: {elapsed_time:.6f}s)')
```

Получим:

Minimum found at: (x=0.271728; y=61.152230) (execution time: 0.000846s)

Как можно заметить, результаты уже сильнее отличаются от найденных ранее.

Решение при помощи scipy

Теперь наследуем еще один класс, но уже для реализации описываемого метода с помощью библиотеки scipy:

```python
# class for scipy implementations
class OptimizationScipy(Optimization):
    # Методравномерногопоиска
    def uniform_search(self, a: float,
                             b: float,
                             n: int = 1000,
                             add_to_comparison: bool = True):
        # start timer
        start_time = time.perf_counter()

        # get result
        res = sp.optimize.brute(self.func, ranges=[(a, b)],
                                Ns=n, full_output=True)
        minpoint = Point(*res[0], res[1])

        # end timer
        result_time = time.perf_counter() - start_time

        # add data to Comparison class
        if add_to_comparison:
            method = Comparison.Method('uniform_search', 'scipy',
                                       minpoint, result_time)
            cmpr.add_data(method)

        return [minpoint, result_time]
```

И запустим:

```python
# create optimizer and find minimum
optimizer = OptimizationScipy(func=myfunc)

min_point, elapsed_time = optimizer.uniform_search(*INTERVAL) # n = 1000 (default)
print(f'Minimum found at: {min_point:.6f} (execution time: {elapsed_time:.6f}s)')
```

Получим:

Minimum found at: (x=0.271009; y=61.152168) (execution time: 0.011910s)

Данные не сильно различаются с теми, что были найдены «самописной» программой. Причем наша программа работает даже быстрее из-за того, что scipy также высчитывает дополнительную информацию (представление оценочной сетки, значения яфункции в каждой точке оценочной сетки).

Решение при помощи PyTorch

Наконец наследуем третий класс для реализации метода на PyTorch:

```
# class for PyTorch implementations
class OptimizationPyTorch(Optimization):
    # Методравномерногопоиска
    def uniform_search(self, a: float,
                             b: float,
                             n: int = 1000,
                             add_to_comparison: bool = True,
                             device='cpu') -> Point:
        # start timer
        start_time = time.perf_counter()

        # one-dimensional grid
        x_tenzor = torch.linspace(start=a, end=b, steps=n, device=device)

        # get corresponding values
        y_tenzor = self.func(x_tenzor)

        # find the index of the minimum value
        min_ind = torch.argmin(y_tenzor)

        # construct a point
        minpoint = Point(x_tenzor[min_ind],
                         y_tenzor[min_ind])

        # end timer
        result_time = time.perf_counter() - start_time

        # add data to Comparison class
        if add_to_comparison:
            method = Comparison.Method('uniform_search', 'pytorch',
                                       minpoint, result_time, device)
            cmpr.add_data(method)

        return [minpoint, result_time]
```

И запустим:

```
# create optimizer and find minimum
optimizer = OptimizationPyTorch(func=myfunc)

# use cpu
min_point, elapsed_time = optimizer.uniform_search(*INTERVAL) # n = 1000 (default)
print(f'(cpu)Minimum found at: {min_point:.6f} (execution time: {elapsed_time:.6f}s)
```

Получим:

(cpu)Minimum found at: (x=0.270270; y=61.152233) (execution time: 0.000442s)

Опять же результат похож на найденные ранее, но при этом работает быстрее. В качестве эксперимента запустим данную реализацию метода при большом значении $n = 10^8$. Причем запустим как на процессоре (cpu), так и на видеокарте (gpu):

```
"""
check for n = 1 * 10^8
"""
# use cpu
min_point, elapsed_time = optimizer.uniform_search(*INTERVAL, n=int(1e8),
                                                   add_to_comparison=False)
print(f'(cpu, n=1e8)Minimum found at: {min_point:.6f} (execution time: {elapsed_time

# use gpu
device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device == 'cpu':
    raise Exception('cuda is not available')

min_point, elapsed_time = optimizer.uniform_search(*INTERVAL, device=device,
                                                   n=int(1e8),
                                                   add_to_comparison=False)
print(f'({'gpu' if device=='cuda' else device}, n=1e8)Minimum found at: {min_point:.
```

Итого имеем:

(cpu, n=1e8)Minimum found at: (x=0.270859; y=61.152168) (execution time: 1.456683s)
(gpu, n=1e8)Minimum found at: (x=0.270858; y=61.152168) (execution time: 0.039908s)

Численно результаты особо не отличаются от предыдущих. Также отметим, что исполнение на видеокарте ожидаемо быстрее.

Метод ломаных

Теория

Аналитическое решение

Реализация метода

Решение при помощи scipy

Решение при помощи PyTorch

Метод секущих

Теория

Аналитическое решение

Реализация метода

Решение при помощи scipy

Решение при помощи PyTorch

Метод Ньютона

Теория

Аналитическое решение

Реализация метода

Решение при помощи scipy

Решение при помощи PyTorch