

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»

Институт технологий

Кафедра инженерной кибернетики

Проектная работа

по дисциплине «Численные методы»

на тему:

«Численные методы решения систем нелинейных уравнений»

Выполнили:
студенты 3-го курса
группы БПМ-21-3
Грачев К.Ю.,
Паршунин В.Н.,
Прокуденко К.И.,
Северин Я.А.,
Люпа Р.А.

Проверил:
старший преподаватель Ширкин С.В.

Москва, 2023

Содержание

Введение	3
Применение численных методов	3
Теоретическая часть	4
Метод простой итерации (метод Якоби)	4
Метод Зейделя	6
Описание технического задания	4
Пользовательская часть	7
Основные требования к модулю	7
Руководство по использованию	8
Техническая часть	8
Реализация модуля Solver	10
Вывод	14

Введение

Решение систем нелинейных уравнений и нахождение корней таких уравнений являются важными задачами в математике и её приложениях. Нелинейные уравнения описывают широкий спектр явлений в науке, технике, экономике и других областях. Решение таких уравнений позволяет понять и предсказать поведение системы, а также найти оптимальные значения параметров.

Однако, решение систем нелинейных уравнений может быть сложной задачей. Нелинейные уравнения могут иметь множество корней или не иметь их вовсе. Кроме того, корни могут быть растянутыми, сложными для анализа или находиться в области, где алгоритмы неэффективны.

В связи с этим, разработка эффективных методов решения систем нелинейных уравнений и поиска их корней является актуальной задачей.

Применение численных методов

Численные методы решения систем нелинейных уравнений находят широкое применение в различных областях, включая:

Наука и инженерия: Численные методы используются для моделирования и анализа сложных физических, химических и технических систем. Например, в электротехнике и электронике, численные методы позволяют анализировать и оптимизировать электрические цепи и электронные устройства.

Экономика и бизнес: Численные методы применяются для решения экономических моделей, оптимизации бизнес-процессов и прогнозирования рыночных трендов. Например, в финансовой аналитике и риск-менеджменте, численные методы используются для моделирования и анализа финансовых инструментов и портфелей.

Медицина и биология: Численные методы применяются для моделирования биологических систем, анализа медицинских данных и прогнозирования заболеваний. Например, в фармакологии и биотехнологии, численные методы используются для

моделирования взаимодействия лекарств с организмами и предсказания эффективности новых препаратов.

Инфраструктура и транспорт: Численные методы применяются для планирования и оптимизации инфраструктуры и транспортных систем. Например, в городском планировании и управлении трафиком, численные методы используются для моделирования потоков транспорта и оптимизации распределения ресурсов.

Искусственный интеллект и машинное обучение: Численные методы являются неотъемлемой частью алгоритмов машинного обучения и искусственного интеллекта. Например, в оптимизации и обучении нейронных сетей, численные методы используются для поиска оптимальных параметров и настройки моделей.

Теоретическая часть

Метод простой итерации (метод Якоби)

Ключевая идея метода Якоби заключается в том, чтобы использовать текущие приближенные значения неизвестных для вычисления новых значений, которые затем используются в следующей итерации. Этот процесс повторяется до тех пор, пока полученные значения не перестанут сильно изменяться.

Продemonстрируем алгоритм метода Якоби. Требуется решить систему нелинейных уравнений вида:

$$\begin{aligned} F_1(x_1, x_2, \dots, x_n) &= 0 \\ F_2(x_1, x_2, \dots, x_n) &= 0 \\ &\dots \\ F_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned} \quad (3.1)$$

Систему нелинейных уравнений (3.1) после преобразований

$$x_i = x_i - \frac{F_i(x)}{M_i}, i = 1, 2, 3, \dots, n$$

(здесь M_i определяются из условия сходимости), представим в виде:

$$x_1 = f_1(x_1, x_2, \dots, x_n)$$

$$x_2 = f_2(x_1, x_2, \dots, x_n) \quad (3.2)$$

...

$$x_n = f_n(x_1, x_2, \dots, x_n)$$

Из системы (3.2) легко получить итерационные формулы метода Якоби. Возьмем в качестве начального приближения какую-нибудь совокупность чисел $x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}$. Подставляя их в правую часть (3.2) вместо переменных x_1, x_2, \dots, x_n , получим новое приближение к решению исходной системы:

$$x_1^{(1)} = f_1(x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$$

$$x_2^{(1)} = f_2(x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}) \quad (3.3)$$

...

$$x_n^{(1)} = f_n(x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$$

Эта операция получения первого приближения $x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}$ решения системы уравнения (3.2) называется первым шагом итерации. Подставляя полученное решение в правую часть уравнения (3.2) получим следующее итерационное приближение: $x_1^{(2)}, x_2^{(2)}, \dots, x_n^{(2)}$ и т.д.:

$$x_i^{(k+1)} = f_i(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}), i = 1, 2, 3, \dots, n \quad (3.4)$$

Итерационный процесс можно считать законченным, если все значения переменных, полученных $(k + 1)$ -ой итерации, отличается от значений соответствующих переменных,

полученных от предыдущей итерации, по модулю меньше наперед заданной точности ε , т.е. если:

$$\max_i |x_i^{(k+1)} - x_i^{(k)}| < \varepsilon \quad (3.5)$$

Пример работы метода Якоби можно представить себе как игру, где каждое уравнение является инструкцией для нахождения значения соответствующей неизвестной. На каждом шаге мы обновляем значения неизвестных согласно этим инструкциям, и игра продолжается до тех пор, пока ответ не станет стабильным.

Метод Зейделя

Алгоритм метода Зейделя состоит из последовательных шагов, которые выполняются до тех пор, пока не будет достигнуто заданное количество итераций или пока разница между двумя последовательными приближениями не станет достаточно мала.

Метод Зейделя отличается от метода Якоби тем, что вычисления ведутся не по формулам (3.4), а по следующим формулам:

$$\begin{aligned} x_1^{(k+1)} &= f_1(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}) \\ x_2^{(k+1)} &= f_2(x_1^{(k+1)}, x_2^{(k)}, \dots, x_n^{(k)}) \\ x_3^{(k+1)} &= f_3(x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_n^{(k)}) \\ &\dots \\ x_n^{(k+1)} &= f_n(x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_n^{(k+1)}) \end{aligned} \quad (3.6)$$

При решении систем нелинейных уравнений необходимо определить приемлемое начальное приближение. Для случая двух уравнений с двумя неизвестными начальное приближение находится графически.

Сходимость метода Зейделя (как и метода Якоби) зависит от вида функции в (3.2), вернее она зависит от матрицы, составленной из частных производных:

$$F = \begin{pmatrix} f_{11} & f_{12} & f_{13} & \cdots & f_{1n} \\ f_{21} & f_{22} & f_{23} & \cdots & f_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_{n1} & f_{n2} & f_{n3} & \cdots & f_{nn} \end{pmatrix} \quad (3.7)$$

$$\text{где } f_{ij} = \frac{\partial f_i}{\partial x_j}$$

Итерационный процесс сходится, если сумма модулей каждой строки F меньше единицы в некоторой окрестности корня:

$$|f_{i1}| + |f_{i2}| + |f_{i3}| + \dots + |f_{in}| < 1, \quad i = 1, 2, 3, \dots, n$$

$$\text{или } \max_{1 \leq i \leq n} \sum_{j=1}^n |f_{ij}| < 1$$

Важно отметить, что метод Зейделя, равно как и метод Якоби, могут быть нестабильными, особенно если начальное приближение выбрано неправильно. Поэтому важно выбирать хорошее начальное приближение и контролировать процесс итерации, чтобы избежать ложных корней или перегрузки.

Описание технического задания

Цель данного проекта – разработка модуля на языке программирования Python для решения систем нелинейных уравнений численными методами. Модуль должен включать в себя реализацию метода простой итерации и метода Зейделя.

Пользовательская часть

Основные требования к модулю

1. Модуль работает безотказно.
2. У пользователя есть возможность не указывать координаты первого приближения.
3. Если первое приближение не было передано в качестве аргумента, оптимальные стартовые координаты подбираются автоматически.
4. Если первое приближение не было передано в качестве аргумента, время сходимости не определено и зависит от функций системы и количества неизвестных в системе.
5. Если первое приближение было передано в качестве аргумента, пользователь получает координаты, в которых метрика показала наилучший результат.

Руководство по использованию

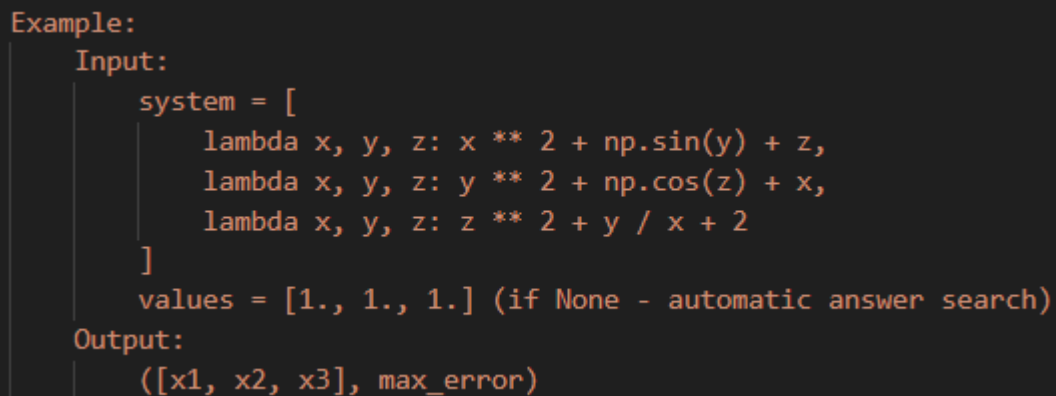
1. Перед использованием необходимо установить переменную среду во избежание конфликта глобальных установленных и локальных используемых модулем версий.

...

```
python -m venv venv
source venv/bin/activate (для Linux)
source venv/scripts/activate (для Windows)
pip install -r requirements.txt
```

...

2. Количество уравнений в системе соответствует количеству неизвестных.
3. При формировании лямбда-выражений, в качестве аргументов должно быть указано количество переменных равное количеству уравнений.
4. Сформируйте список из лямбда-выражений. К примеру:



```
Example:
Input:
    system = [
        lambda x, y, z: x ** 2 + np.sin(y) + z,
        lambda x, y, z: y ** 2 + np.cos(z) + x,
        lambda x, y, z: z ** 2 + y / x + 2
    ]
    values = [1., 1., 1.] (if None - automatic answer search)
Output:
    ([x1, x2, x3], max_error)
```

5. Сформируйте список координат начального приближения по желанию.
6. В качестве обязательного аргумента передайте сформированный список, представляющий собой систему уравнений вида $F_i(x_1, x_2, \dots, x_n) = 0$, а также список координат начального приближения, степень точности (eps) и параметр сходимости (alpha) по желанию.

Техническая часть

Класс Solver

Solver - это класс, предоставляющий реализацию нескольких методов решения систем нелинейных уравнений. В данном коде реализован метод простой итерации (метод Якоби).

Методы класса Solver

1. `__max_error(system: list, args: list) -> float`

Этот статический метод вычисляет максимальную ошибку в системе уравнений для заданных аргументов.

2. `__wrapper(lambda_function) -> 'function'`

Этот статический метод создает обертку для удобства использования лямбда-функций.

3. `Jacobi(cls, system: list, values=None, eps=1e-4) -> (list, float)`

Этот метод реализует метод простой итерации (метод Якоби) для решения системы нелинейных уравнений.

system: список лямбда-функций, представляющих систему уравнений.

values: начальные значения переменных. Если None, будет произведена попытка поиска оптимальной начальной точки.

alpha: параметр сходимости. Если ошибка неудовлетворительна, необходимо изменить этот параметр.

eps: критерий остановки (по умолчанию $1e-4$).

3.1. Внутренние функции метода Jacobi

1. `epoch(values: np.ndarray) -> (list, float)`

Эта функция представляет собой реализацию одного полного цикла схождения метода простой итерации (метода Якоби).

2. Основной цикл

*В основном цикле метода Якоби выполняется процесс итераций до достижения критерия остановки (*eps*) или максимального числа итераций (в данном случае 1000).*

3. Поиск оптимальной начальной точки

*Если *values* не задан, метод производит попытки найти оптимальную начальную точку, изменяя начальные значения переменных и сохраняя наилучший результат.*

4. Seidel(cls, system: list, values=None, alpha = 0.01, eps=1e-4) -> (list, float)

Этот метод реализует метод Зейделя для решения системы нелинейных уравнений.

system: список лямбда-функций, представляющих систему уравнений.

values: начальные значения переменных. Если None, будет произведена попытка поиска оптимальной начальной точки.

alpha: параметр сходимости. Если ошибка неудовлетворительна, необходимо изменить этот параметр.

eps: критерий остановки (по умолчанию 1e-4).

4.1. Внутренние функции метода Seidel

1. epoch(values: np.ndarray) -> (list, float)

Эта функция представляет собой реализацию одного полного цикла схождения метода Зейделя.

2. Основной цикл

В основном цикле метода Seidel выполняется процесс итераций до достижения критерия остановки (eps) или максимального числа итераций (в данном случае 1000).

3. Поиск оптимальной начальной точки

Если values не задан, метод производит попытки найти оптимальную начальную точку, изменяя начальные значения переменных и сохраняя наилучший результат.

Обработка ошибок

В коде предусмотрена обработка различных ошибок, таких как NaN или inf значения в процессе вычислений, а также проверка на достижение экстремально больших значений.

Реализация модуля Solver

```
import numpy as np
from copy import deepcopy
import warnings
warnings.simplefilter('ignore')
```

```

class Solver:
    def __init__():
        pass

    @staticmethod # Metric
    def __max_error(system: list, args: list) -> float:
        return max([abs(lam(*args)) for lam in system])

    @staticmethod # For comfortable lambdas exploitation
    def __wrapper(lambda_function) -> 'function':
        def wrapped_function(args: list):
            return lambda_function(*args)

        return wrapped_function

    @classmethod # The simple iteration method
    def Jacobi(cls, system: list, values=None, alpha=0.01, eps=1e-4) -> (list,
float):
        ...
        The simple iteration method (Jacobi method).
        Example:
        Input:
            system = [
                lambda x, y, z: x ** 2 + np.sin(y) + z,
                lambda x, y, z: y ** 2 + np.cos(z) + x,
                lambda x, y, z: z ** 2 + y / x + 2
            ]
            values = [1., 1., 1.] (if None - automatic answer search)
        Output:
            ([x1, x2, x3], max_error)
        ...

    def epoch(values: np.ndarray): # Method implementation
        answer = []
        min_error = np.inf
        error_last = np.inf
        iter = 0
        while error_last > eps and iter < 1000:
            iter += 1
            new_values = np.array([round(values[i] - wsystem[i](values) *
alpha, 5) for i in range(n)]) # Calculating new values at i-step
            if any(np.isnan(new_values)) or any(np.isinf(new_values)) or
np.any(new_values[abs(new_values) > 1e10]): # Avoiding problems
                break
            error_cur = Solver.__max_error(system, new_values)
            if error_cur == np.inf:
                break
            if error_cur < min_error:
                answer, min_error = new_values, error_cur
            if max(np.absolute(new_values - values)) < eps:

```

```

        break
        error_last = error_cur
        values = new_values

    if len(answer):
        return answer, min_error

    return values, error_last

n = len(system)
wsystem = [Solver.__wrapper(lam) for lam in system]
if values is not None: # The result for the proposed first approximation
will be returned
    if not isinstance(values, np.ndarray):
        values = np.array(values)
    return epoch(values)

answer = []
min_error = np.inf
for step_rate in [0.025, 0.1]: # Attempts to find the optimal starting
point
    start_rate = step_rate
    start = np.ones((n, )) + eps
    for i in range(1000):
        if not i % 20:
            step_rate += (start_rate - (start_rate / 5) * (i // 1000))
        err = epoch(start)[1]
        for j in range(n):
            sign = 1
            iter = 1
            while True:
                if not iter % 10:
                    break
                start[j] += sign * step_rate
                ans, cur_err = epoch(start)
                if cur_err < eps:
                    return ans, cur_err
                elif cur_err < min_error:
                    answer, min_error = ans, cur_err
                if cur_err < err:
                    err = cur_err
                else:
                    if iter == 1 and cur_err > err:
                        sign *= -1
                        start[j] += sign * step_rate
                    elif iter != 1 and cur_err >= err:
                        break
                iter += 1

    return answer, min_error

```

```

    @classmethod # iteration method with forward substitution
    def Seidel(cls, system: list, values=None, alpha=0.01, eps=1e-4) -> (list,
float):
    ...
        Iteration method with forward substitution (Seidel method).
        Example:
        Input:
            system = [
                lambda x, y, z: x ** 2 + np.sin(y) + z,
                lambda x, y, z: y ** 2 + np.cos(z) + x,
                lambda x, y, z: z ** 2 + y / x + 2
            ]
            values = [1., 1., 1.] (if None - automatic answer search)
        Output:
            ([x1, x2, x3], max_error)
    ...

def epoch(values: np.ndarray): # Method implementation
    answer = []
    min_error = np.inf
    error_last = np.inf
    iter = 0
    while error_last > eps and iter < 1000:
        iter += 1
        new_values = deepcopy(values)
        new_values[0] = round(values[0] - wsystem[0](values) * alpha, 5)
        for i in range(1, n):
            new_values[i] = round(values[i] - wsystem[i](new_values) *
alpha, 5)
            if any(np.isnan(new_values)) or any(np.isinf(new_values)) or
np.any(new_values[abs(new_values) > 1e10]): # Avoiding problems
                break
            error_cur = Solver.__max_error(system, new_values)
            if error_cur == np.inf:
                break
            if error_cur < min_error:
                answer, min_error = new_values, error_cur
            if max(np.absolute(new_values - values)) < eps:
                break
            error_last = error_cur
            values = new_values

    if len(answer):
        return answer, min_error

    return values, error_last

n = len(system)
wsystem = [Solver.__wrapper(lam) for lam in system]

```

```

        if values is not None: # The result for the proposed first approximation
will be returned
            if not isinstance(values, np.ndarray):
                values = np.array(values)
            return epoch(values)

    answer = []
    min_error = np.inf
    for step_rate in [0.025, 0.1]: # Attempts to find the optimal starting
point
        start_rate = step_rate
        start = np.ones((n, )) + eps
        for i in range(1000):
            if not i % 20:
                step_rate += (start_rate - (start_rate / 5) * (i // 1000))
            err = epoch(start)[1]
            for j in range(n):
                sign = 1
                iter = 1
                while True:
                    if not iter % 10:
                        break
                    start[j] += sign * step_rate
                    ans, cur_err = epoch(start)
                    if cur_err < eps:
                        return ans, cur_err
                    elif cur_err < min_error:
                        answer, min_error = ans, cur_err
                    if cur_err < err:
                        err = cur_err
                    else:
                        if iter == 1 and cur_err > err:
                            sign *= -1
                            start[j] += sign * step_rate
                        elif iter != 1 and cur_err >= err:
                            break
                    iter += 1

        return answer, min_error

```

Вывод

В представленной работе был реализован модуль из двух методов для численного решения систем нелинейных уравнений. Оба численных метода конкурируют друг с другом в точности и скорости в разных задачах. Стоит отметить, что предпочтение было отдано не

скорости работы, а точности, но задел под оптимизацию был предусмотрен. В отчете также представлены техническая и пользовательская документации для лучшего понимания и комфортного эксплуатирования модуля.