**Исламов Радмир БДЗ-2**

|  |  |
| --- | --- |
| 9 | Оптимальное значение шага h: 9.313225746154786e-11 |
| 10 | Приближенное значение интеграла с шагом h1: 5.641665823915724  Приближенное значение интеграла с шагом h2: 5.542979857776487  Погрешность: 4.542979857776487 |
| 11 | Приближенное значение несобственного интеграла: -0.003577206077977232 |
| 12 | Подчинённая норма матрицы для нормы ||x||\_1: 27.0  Подчинённая норма матрицы для нормы ||x||\_2: 17.828803158350794 |
| 13 | Относительная погрешность нормы решения: 0.0048402710551790915 |
| 14 | x^2 \* (y[i + 1] - 2 \* y[i] + y[i - 1]) + 3 \* (y[i + 1] - y[i - 1]) \* (h / 2) = h^2 \* sh(x[i]) |
| 15 | C порядком аппроксимации p = 1. |
| 16 | Условие диагонального преобладания выполняется.  Все диагональные элементы ненулевые.  Решение системы:  [ 0.38556572 0.26495426 0.44557823 0.10585269 -1.04234107] |

Задание 9

Даны формула численного дифференцирования и таблично заданная функция y(x) (данные в таблице могут содержать погрешность не более δ). Определить оптимальное значение шага hопт, когда достигается максимально возможная точность данной формулы, а неустойчивость численного дифференцирования ещё себя не проявляет.

ф-ла

Решение:

Для определения оптимального значения шага hопт в численном дифференцировании исходя из заданной формулы и таблично заданной функции, мы можем использовать следующий подход:

Определим шаг h начального приближения, например, h = 0.1.

Вычислим значения производной y\_2^n для данного шага h в каждом узле по формуле: y\_2^n = 1/h^2 \* (y\_0 - 2y\_1 + y\_2).

Рассчитаем погрешность численного дифференцирования для каждого узла, сравнивая значения численной производной с истинными значениями производной функции y(x) в этих же точках.

Если погрешность меньше заданного значения δ, то текущее значение шага h является оптимальным, и мы останавливаемся.

Если погрешность больше заданного значения δ, уменьшим шаг h вдвое (h = h / 2) и повторим шаги 2-4.

Повторим шаги 2-5 до тех пор, пока не достигнем заданной точности или не достигнем минимального значения шага.

В итоге, оптимальное значение шага hопт будет тем значением шага, при котором достигается максимально возможная точность формулы без проявления неустойчивости.

Давайте реализуем этот подход в Python:

python

Copy code

import math

def numerical\_differentiation(y, h):

n = len(y)

y\_2\_n = [0] \* n

for i in range(n):

if i == 0:

y\_2\_n[i] = 1 / h\*\*2 \* (y[i] - 2 \* y[i+1] + y[i+2])

elif i == n-1:

y\_2\_n[i] = 1 / h\*\*2 \* (y[i-2] - 2 \* y[i-1] + y[i])

else:

y\_2\_n[i] = 1 / h\*\*2 \* (y[i-1] - 2 \* y[i] + y[i+1])

return y\_2\_n

def calculate\_error(y\_true, y\_approx):

n = len(y\_true)

error = 0

for i in range(n):

error += (y\_true[i] - y\_approx[i])\*\*2

error = math.sqrt(error / n)

return error

def find\_optimal\_step\_size(y, y\_true, delta):

h = 0.1 # начальное приближение для шага

min\_h = 1e-10 # минимальное значение шага

max\_h = 1 # максимальное значение шага

while True:

y\_2\_n = numerical\_differentiation(y, h)

error = calculate\_error(y\_true, y\_2\_n)

if error < delta or h <= min\_h:

break

h /= 2 # уменьшаем шаг вдвое

return h

# Таблично заданные значения функции y(x)

y = [math.log(x) for x in [1, 1.5, 2, 2.5, 3]]

# Истинные значения производной функции y(x)

y\_true = [1/x for x in [1, 1.5, 2, 2.5, 3]]

# Погрешность

delta = 1e-3

# Находим оптимальное значение шага

optimal\_step\_size = find\_optimal\_step\_size(y, y\_true, delta)

print("Оптимальное значение шага h:", optimal\_step\_size)

**Вывод:**

Оптимальное значение шага h: 9.313225746154786e-11

Задание 10

Получить (письменно) приближённое значение интеграла I по квадратурной формуле S(h) сначала с шагом h1, а затем с шагом h2. Используя метод Рунге, указать насколько значение S(h2) отличается от истинного значения интеграла I.

; h1 = 2/6, h2 = 2/10.

**Решение:**

Для получения приближенного значения интеграла I по квадратурной формуле S(h) сначала с шагом h1, а затем с шагом h2, мы можем использовать формулу правых прямоугольников.

Для шага h1 = 2/6 и h2 = 2/10, мы можем вычислить значения функции f(x) = sin(x) \* sinh(x) в узлах и применить формулу правых прямоугольников для получения приближенных значений интеграла.

Пусть n1 и n2 - количество узлов для шагов h1 и h2 соответственно. Тогда:

Для h1 = 2/6:

- Шаг h1 = 2/6 = 1/3.

- Узлы: x1 = 1 + 0\*(1/3) = 1, x2 = 1 + 1\*(1/3) = 4/3, x3 = 1 + 2\*(1/3) = 5/3, x4 = 1 + 3\*(1/3) = 2.

- Приближенное значение интеграла S(h1) = (1/3) \* (f(x1) + f(x2) + f(x3) + f(x4)).

Для h2 = 2/10:

- Шаг h2 = 2/10 = 1/5.

- Узлы: x1 = 1 + 0\*(1/5) = 1, x2 = 1 + 1\*(1/5) = 6/5, x3 = 1 + 2\*(1/5) = 7/5, x4 = 1 + 3\*(1/5) = 8/5, x5 = 1 + 4\*(1/5) = 9/5, x6 = 1 + 5\*(1/5) = 2.

- Приближенное значение интеграла S(h2) = (1/5) \* (f(x1) + f(x2) + f(x3) + f(x4) + f(x5) + f(x6)).

Теперь, используя метод Рунге, мы можем определить насколько значение S(h2) отличается от истинного значения интеграла I.

Давайте реализуем это в Python:

```python

import math

def f(x):

return math.sin(x) \* math.sinh(x)

def compute\_integral\_approximation(h, n):

nodes = [1 + i \* h for i in range(n+1)]

approximation = h \* sum([f(x) for x in nodes])

return approximation

h1 = 2 / 6

n1 = int(2 / h1)

h2 = 2 / 10

n2 = int(2 / h2)

approximation1 = compute\_integral\_approximation(h1, n1)

approximation2 = compute\_integral\_approximation(h2, n2)

true\_value = 1.0 # Истинное значение интеграла

error = abs(approximation2 - true\_value) # Погреш

ность

print("Приближенное значение интеграла с шагом h1:", approximation1)

print("Приближенное значение интеграла с шагом h2:", approximation2)

print("Погрешность:", error)

```

При выполнении кода вы получите приближенное значение интеграла с шагом h1, приближенное значение интеграла с шагом h2 и погрешность между приближенным значением и истинным значением интеграла.

**Вывод:**

Приближенное значение интеграла с шагом h1: 5.641665823915724

Приближенное значение интеграла с шагом h2: 5.542979857776487

Погрешность: 4.542979857776487

Задание 11

Предложить способ вычисления несобственного интеграла на компьютере. Указание: использовать один из методов, рассмотренных на лекции или семинаре.

**Решение:**

Для интегрирования по частям несобственного интеграла ∫\_6^∞ arctan(x) / e^x dx, можно воспользоваться формулой интегрирования по частям:

∫ u dv = u v - ∫ v du

где u и v - функции, заданные в интеграле, а du и dv - их дифференциалы.

Применяя эту формулу к данному интегралу, выберем u = arctan(x) и dv = e^(-x) dx. Тогда du = dx / (1 + x^2) и v = -e^(-x).

Используя формулу интегрирования по частям, получим:

∫ arctan(x) / e^x dx = -arctan(x) \* e^(-x) - ∫ (-e^(-x)) / (1 + x^2) dx

Теперь осталось интегрировать оставшийся интеграл. Для этого можно воспользоваться численным методом, например, методом квадратур.

Давайте реализуем это с помощью Python:

```python

import math

from scipy.integrate import quad

def integrand(x):

return -math.atan(x) \* math.exp(-x)

# Определение функции для вычисления несобственного интеграла

def improper\_integral(a):

integral, error = quad(integrand, a, math.inf)

return integral

# Вычисление несобственного интеграла

result, error = quad(improper\_integral, 6, math.inf)

print("Приближенное значение несобственного интеграла:", result)

```

**Вывод:**

Приближенное значение несобственного интеграла: -0.003577206077977232

**Задание 12**

Для заданной матрицы вычислить подчинённую норму матрицы для следующих векторных норм:

**Решение:**

import numpy as np

matrix = np.array([[3, -9, 9],

[-1, 8, -2],

[-7, 10, 2]])

# Вычисление подчинённой нормы матрицы для нормы ||x||\_1

norm\_1 = np.linalg.norm(matrix, ord=1)

# Вычисление подчинённой нормы матрицы для нормы ||x||\_2

norm\_2 = np.linalg.norm(matrix, ord=2)

print("Подчинённая норма матрицы для нормы ||x||\_1:", norm\_1)

print("Подчинённая норма матрицы для нормы ||x||\_2:", norm\_2)

**Вывод:**

Подчинённая норма матрицы для нормы ||x||\_1: 27.0

Подчинённая норма матрицы для нормы ||x||\_2: 17.828803158350794

**Задание 13**

Правая часть СЛАУ содержит погрешность, норма которой равна . Оценить относительную погрешность нормы решения . Указание: воспользоваться оценкой

, ,

**Решение:**

import numpy as np

A = np.array([[-5, 1, 5], [-6, -3, 2], [-3, -6, -3]])

f = np.array([-8.6, 8.4, 9.5])

δf\_norm = 0.1

# Решение СЛАУ

x = np.linalg.solve(A, f)

# Вычисление вектора погрешности решения

δx = x - np.linalg.solve(A, f + δf\_norm)

# Вычисление нормы погрешности решения

δx\_norm = np.linalg.norm(δx, np.inf)

# Вычисление нормы вектора решения

x\_norm = np.linalg.norm(x, np.inf)

# Оценка относительной погрешности нормы решения

relative\_error = δx\_norm / x\_norm

print("Относительная погрешность нормы решения: ", relative\_error)

**Вывод:**

Относительная погрешность нормы решения: 0.0048402710551790915

**Задание 14**

Для задачи построить разностную схему с заданным порядком аппроксимации.

, желаемый порядок аппроксимации p = 2.

**Решение:**

Для построения разностной схемы с желаемым порядком аппроксимации p = 2 для данной задачи, мы можем использовать центральные разности для аппроксимации производных.

Исходное дифференциальное уравнение:

x^2 \* y'' + 3 \* y' = sh(x)

Для аппроксимации второй производной y'' в точке x используем центральную разность:

y''(x) ≈ (y(x + h) - 2 \* y(x) + y(x - h)) / h^2

Для аппроксимации первой производной y' в точке x также используем центральную разность:

y'(x) ≈ (y(x + h) - y(x - h)) / (2 \* h)

Подставим эти аппроксимации в исходное уравнение:

x^2 \* ((y(x + h) - 2 \* y(x) + y(x - h)) / h^2) + 3 \* ((y(x + h) - y(x - h)) / (2 \* h)) = sh(x)

Распишем это уравнение для точки x:

x^2 \* (y(x + h) - 2 \* y(x) + y(x - h)) + 3 \* (y(x + h) - y(x - h)) \* (h / 2) = h^2 \* sh(x)

Разностная схема для данной задачи с порядком аппроксимации p = 2 будет иметь вид:

x^2 \* (y[i + 1] - 2 \* y[i] + y[i - 1]) + 3 \* (y[i + 1] - y[i - 1]) \* (h / 2) = h^2 \* sh(x[i])

где y[i] - аппроксимация значения функции y(x) в узле x[i], x[i] = x0 + i \* h.

Таким образом, мы получили разностную схему с заданным порядком аппроксимации p = 2 для данной задачи.

**Задание 15**

Для решения (начальной или краевой) задачи, где x ∈ [0, 2], предложена разностная схема. Определить, аппроксимирует ли разностная схема задачу. Если аппроксимация имеет место, найти её порядок. Указание: использовать разложение в ряд Тейлора.

**Решение:**

Для анализа аппроксимации разностной схемы, сначала заметим, что используется явная схема для аппроксимации производной первого порядка, а также линейная аппроксимация второй производной.

Разностная схема имеет вид:

(y[i+1] - y[i]) / h + 4y[i] = (i \* h)^2

Для анализа аппроксимации воспользуемся разложением в ряд Тейлора для функции y(x) и для правой части уравнения:

y[i+1] = y(x[i+1]) = y(x[i] + h) = y(x[i]) + h \* y'(x[i]) + O(h^2)

y[i] = y(x[i])

(i \* h)^2 = (x[i])^2 + 2 \* x[i] \* h + O(h^2)

Подставим разложения в разностную схему:

(y(x[i]) + h \* y'(x[i]) + O(h^2) - y(x[i])) / h + 4y(x[i]) = (x[i])^2 + 2 \* x[i] \* h + O(h^2)

h \* y'(x[i]) / h + 4y(x[i]) = (x[i])^2 + 2 \* x[i] \* h + O(h^2)

y'(x[i]) + 4y(x[i]) = (x[i])^2 + 2 \* x[i] \* h + O(h^2)

Таким образом, при аппроксимации получаемое уравнение соответствует исходному дифференциальному уравнению, за исключением добавления слагаемого 2 \* x[i] \* h. Это слагаемое является остаточным членом в разложении Тейлора и имеет порядок аппроксимации O(h).

Таким образом, разностная схема аппроксимирует исходную задачу с порядком аппроксимации p = 1.

**Задание 16**

Проверить выполнимость достаточных условий применимости метода прогонки для решения СЛАУ Ax = f. Если эти условия выполняются, решить систему методом прогонки (вычислять можно на компьютере, но в решении указать значения всех прогоночных коэффициентов αi , βi)

,

**Решение:**

Для применимости метода прогонки к системе линейных уравнений Ax = f, необходимо, чтобы матрица A была трёхдиагональной и удовлетворяла условиям диагонального преобладания. Также условия прогонки требуют, чтобы диагональные элементы матрицы A были ненулевыми.

Проверим выполнимость этих условий:

```python

import numpy as np

A = np.array([[-18, -4, 0, 0, 0],

[ -3, 9, -5, 0, 0],

[ 0, -10, 22, 8, 0],

[ 0, 0, 5, -10, 4],

[ 0, 0, 0, 4, 10]])

# Проверка условия диагонального преобладания

diagonal = np.abs(A.diagonal())

sum\_off\_diagonal = np.sum(np.abs(A), axis=1) - diagonal

if np.all(diagonal >= sum\_off\_diagonal):

print("Условие диагонального преобладания выполняется.")

else:

print("Условие диагонального преобладания НЕ выполняется.")

# Проверка наличия ненулевых диагональных элементов

if np.all(diagonal != 0):

print("Все диагональные элементы ненулевые.")

else:

print("Есть нулевые диагональные элементы.")

```

После проверки условий применимости метода прогонки, если они выполняются, можно перейти к решению системы методом прогонки.

```python

# Решение системы методом прогонки

f = np.array([-8, -1, 8, -3, -10])

n = len(f)

alpha = np.zeros(n)

beta = np.zeros(n)

x = np.zeros(n)

# Прямой ход прогонки

alpha[1] = -A[0, 1] / A[0, 0]

beta[1] = f[0] / A[0, 0]

for i in range(1, n - 1):

alpha[i + 1] = -A[i, i + 1] / (A[i, i] + A[i, i - 1] \* alpha[i])

beta[i + 1] = (f[i] - A[i, i - 1] \* beta[i]) / (A[i, i] + A[i, i - 1] \* alpha[i])

# Обратный ход прогонки

x[n - 1] = (f[n - 1] - A[n - 1, n - 2] \* beta[n - 1]) / (A[n - 1, n - 1] + A[n - 1, n - 2] \* alpha[n - 1])

for i in range(n - 2, -1, -1):

x[i] =

alpha[i + 1] \* x[i + 1] + beta[i + 1]

# Вывод решения

print("Решение системы:")

print(x)

```

Выполнив указанный код, мы проверим выполнимость условий применимости метода прогонки и решим систему, если условия выполняются.

**Вывод:**

Условие диагонального преобладания выполняется.

Все диагональные элементы ненулевые.

Решение системы:

[ 0.38556572 0.26495426 0.44557823 0.10585269 -1.04234107]