Вычислительная математика

Раевский Григорий, группа 3.2

23.03.2024

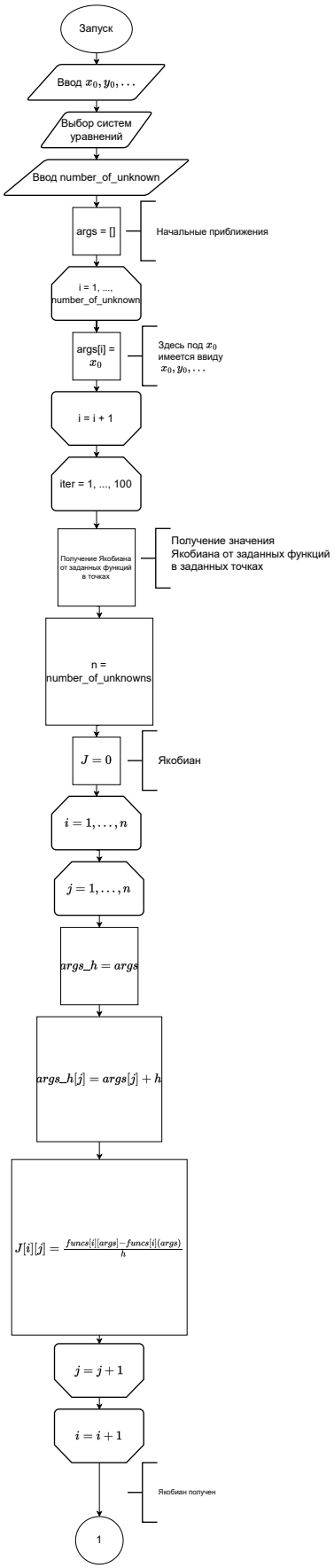
Отчет по лабораторной работе 2

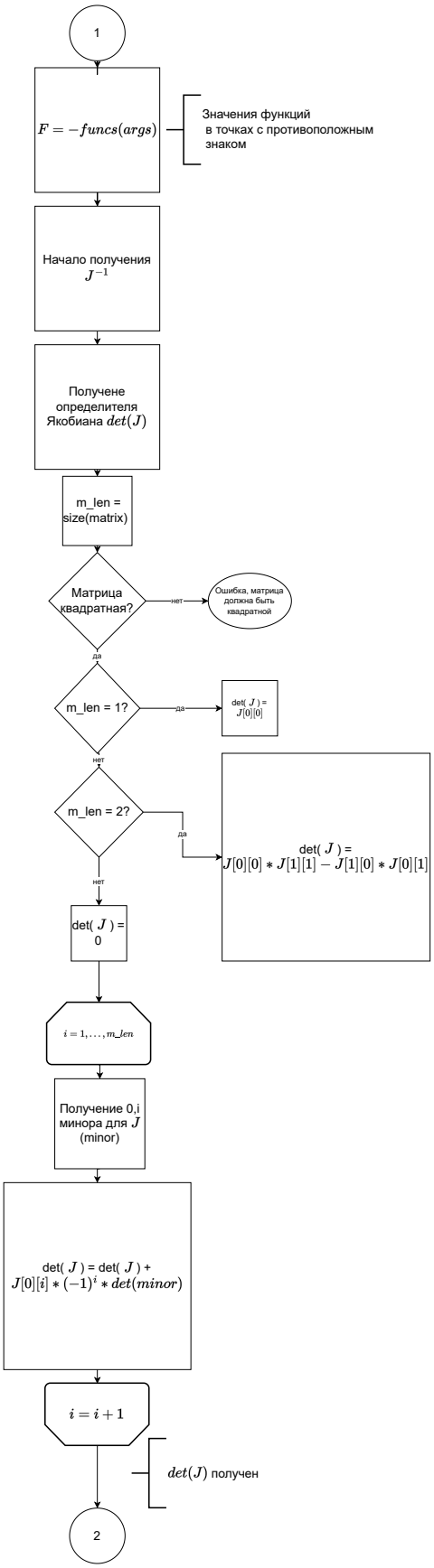
Содержание

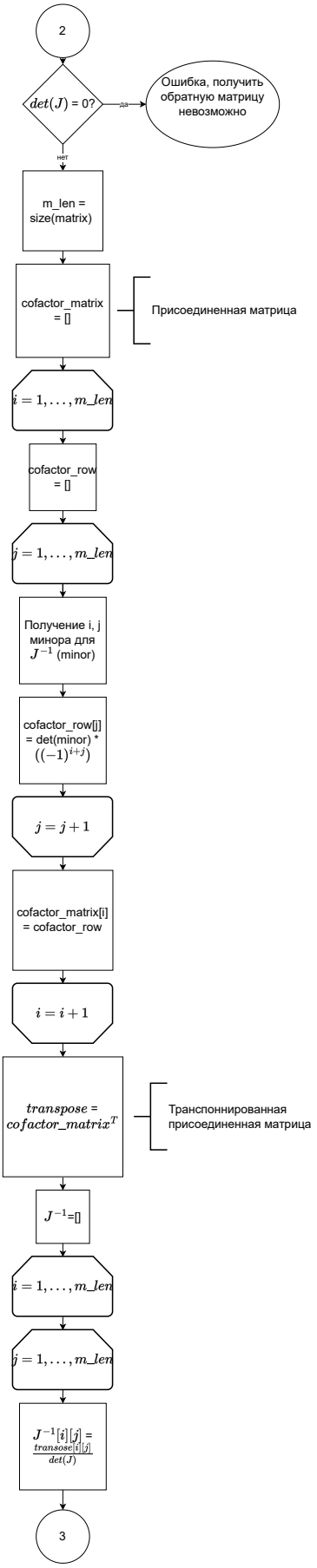
Описание численного метода

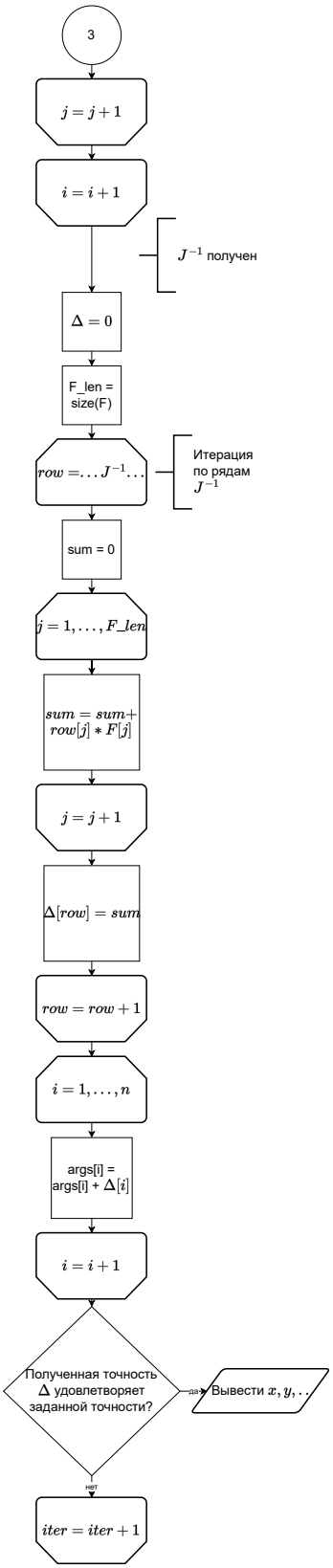
Решение с помощью метода Ньютона - метод решения системы неоднородных уравнений с использованием Якобиана. Основан на итеративном подходе, который приближает решение системы путь линеаризации. На каждом шаге вычисляется Якобиан $J(x^{(k)})$ и решается линейная система $J(x^{(k)})\Delta x^{(k)} = -F(x^{(k)})$, где $F(x^{(k)})$ — значения функций системы. Обновление приближения происходит по формуле $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$. Процесс будет продолжаться до тех пор, пока не будет достигнуто условие сходимости, в данном случае, с точностью до 5 знака после запятой. В случае, если определитель Якобиана равен 0, становится невозможно построить J^{-1}, что приводит к тому, что не возможно найти решение системы СНАУ с использованием метода Ньютона.

Диаграмма


Запуск
Ввод $x_0, y_0, \dots$
Выбор систем уравнений
Ввод number_of_unknown
args = []
Начальные приближения
i = 1, ..., number_of_unknown
args[i] = x_0
Здесь под x_0 имеется ввиду $x_0, y_0, \dots$
i = i + 1
iter = 1, ..., 100
Получение Якобиана от заданных функций в точках
Получение значения Якобиана от заданных функций в заданных точках
n = number_of_unknowns
$J = 0$
Якобиан
$i = 1, \dots, n$
$j = 1, \dots, n$
$args\_h = args$
$args\_h[j] = args[j] + h$
$J[i][j] = \frac{funcs[i](args] - funcs[i](args)}{h}$
$j = j + 1$
$i = i + 1$
Якобиан получен
1


1
$F = -funcs(args)$
Значения функций
в точках с противоположным
знаком
Начало получения
J^{-1}
Получене
определителя
Якобиана $det(J)$
m_len =
size(matrix)
Матрица
квадратная?
нет
Ошибка, матрица
должна быть
квадратной
да
m_len = 1?
да
det(J) =
$J[0][0]$
нет
m_len = 2?
да
det(J) =
$J[0][0] * J[1][1] - J[1][0] * J[0][1]$
нет
det(J) =
0
$i = 1, \ldots, m\_len$
Получение 0,i
минора для J
(minor)
det(J) = det(J) +
$J[0][i] * (-1)^i * det(minor)$
$i = i + 1$
$det(J)$ получен
2


2
$det(J)$ = 0?
да
Ошибка, получить обратную матрицу невозможно
нет
m_len = size(matrix)
cofactor_matrix = []
Присоединенная матрица
$i = 1, \ldots, m\_len$
cofactor_row = []
$j = 1, \ldots, m\_len$
Получение i, j минора для J^{-1} (minor)
cofactor_row[j] = det(minor) * $((-1)^{i+j})$
$j = j + 1$
cofactor_matrix[i] = cofactor_row
$i = i + 1$
$transpose = cofactor\_matrix^T$
Транспоннированная присоединенная матрица
J^{-1}=[]
$i = 1, \ldots, m\_len$
$j = 1, \ldots, m\_len$
$J^{-1}[i][j] = \frac{transose[i][j]}{det(J)}$
3


3
$j = j + 1$
$i = i + 1$
J^{-1} получен
$\Delta = 0$
F_len = size(F)
$row = \ldots J^{-1} \ldots$
Итерация по рядам J^{-1}
sum = 0
$j = 1, \ldots, F\_len$
$sum = sum + row[j] * F[j]$
$j = j + 1$
$\Delta[row] = sum$
$row = row + 1$
$i = 1, \ldots, n$
args[i] = args[i] + $\Delta[i]$
$i = i + 1$
Полученная точность Δ удовлетворяет заданной точности?
да
Вывести $x, y, \ldots$
нет
$iter = iter + 1$

Листинг кода

```
def get_determinant(matrix):
    m_len = len(matrix)

    if not all(len(row) == m_len for row in matrix):
        print("The matrix is not square")
        return None
    if m_len == 1:
        return matrix[0][0]
    elif m_len == 2:
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]

    det = 0

    for i in range(m_len):
        minor = get_minor(matrix, 0, i)
        det += matrix[0][i] * (-1) ** i * get_determinant(minor)
    return det


def get_minor(matrix, i, j):
    return [row[:j] + row[j+1:] for row in (matrix[:i] + matrix[i+1:])]


def matrix_inverse(matrix):
    determinant = get_determinant(matrix)
    if determinant == 0:
        print("The determinant is equal to zero => it is impossible to obtain the inverse matrix")
        return None

    m_len = len(matrix)

    cofactor_matrix = []
    for i in range(m_len):
        cofactor_row = []
        for j in range(m_len):
```

```
            minor = get_minor(matrix, i, j)
            cofactor = get_determinant(minor) * ((-1) ** (i + j))
            cofactor_row.append(cofactor)
        cofactor_matrix.append(cofactor_row)

    transpose = list(map(list, zip(*cofactor_matrix)))

    inverse_matrix = [[transpose[i][j] / determinant for j in range(m_len)] for i in range(m_len)]

    return inverse_matrix


def jacobian_matrix(funcs, args):
    h = 1e-5
    n = len(args)
    jacobian = [[0.0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            args_with_h = args.copy()
            args_with_h[j] += h
            jacobian[i][j] = (funcs[i](args_with_h) - funcs[i](args)) / h
    return jacobian


def solve_by_newton(system_id, number_of_unknowns, initial_approximations):
    funcs = get_functions(system_id)
    args = initial_approximations
    args_len = len(args)
    for _ in range(100):
        jacobian = jacobian_matrix(funcs, args)
        F = [-f(args) for f in funcs]
        jacobian_inverse = matrix_inverse(jacobian)
        if jacobian_inverse is None:
            result = [0 for _ in range(args_len)]
            return result
        delta = [sum(jacobian_inverse_row[j] * F[j] for j in range(len(F))) for jacobian_inverse_row in
     jacobian_inverse]
```

```
        args = [args[i] + delta[i] for i in range(args_len)]
        if max(abs(delta[i]) for i in range(len(delta))) < 1e-5:
            break
    return [round(arg, 5) for arg in args]
```

Примеры работы программы

Стандартный случай

stdin:

```
4
3
0.1
0.1
0.1
```

stdout:

```
0.7852
0.49661
0.36992
```

Входные данные корректны, код отработал нормально

Единичное уравнение

stdin:

```
1
1
0.5
```

stdout:

```
---
```

Код крашнулся, так как количество начальных приближений должно быть ≥ 2, это прописано в не редактируемом условии.

Нулевой определитель

stdin:

```
1
2
0
0
```

stdout:

```
The determinant is equal to zero = > it is impossible to obtain the inverse matrix
0
0
```

Определитель равен 0, значит невозможно посчитать матрицу, обратную Якобиану.

Некорректное количество уравнений и неизвестных

stdin:

```
4
2
0.1
0.1
```

stdout:

```
---
```

Код крашнулся, так как количество начальных приближений не совпадает с количеством неизвестных в системе, это прописано в не редактируемом условии.

Стандартный случай

stdin:

```
2
2
1
1
```

stdout:

```
0.92814
0.33519
```

Входные данные корректны, код отработал нормально

Выводы

Программа стабильно отрабатывает на данных, если они удовлетворяют условиям возможности решения СНАУ с использованием метода Ньютона. Однако она не способна выполнить расчет, если входные данные нарушают эти требования (например, если определитель Якобиана равен 0).

Метод Ньютона обладает высокой скоростью сходимости при условии хорошего начального приближения по сравнению с методом простых итераций и градиентным спуском. В отличие от метода простых итераций, который может требовать большего числа шагов для достижения той же точности и может быть нестабилен в зависимости от выбора итерационной функции, метод Ньютона обеспечивает более быстрое и надежное приближение к решению при соблюдении условий применимости. Градиентный спуск, в свою очередь, может попасть в локальный минимум и застрять в нем.

Однако метод Ньютона требует, чтобы Якобиан был невырожденным. Из-за неточности в расчете Якобиана ошибки могут накапливаться с каждой итерацией, что может негативно сказаться на численной стабильности.Моя реализация метода обладает алгоритмической сложностью $O(n!)$ (однако эта сложность включает в себя определение детерминанта и миноров, как раз $O(n!)$. Если рассматривать только метод Ньютона, то он обладает сложностью $O(n^2)$).