**Министерство науки и высшего образования Российской Федерации**

федеральное государственное автономное образовательное учреждение высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**Отчет**

по лабораторной работе №3

по дисциплине «**Вычислительная математика**»

Автор: Баянов Равиль Динарович

Факультет: ПИиКТ

Группа: P3234

Преподаватель: Перл О. В.

Изображение выглядит как черный, темнота

Автоматически созданное описание

Санкт-Петербург, 2024

Оглавление

[Описание метода 3](#_Toc160999556)

[Блок-схема 4](#_Toc160999557)

[Исходный код метода на языке программирования Python 5](#_Toc160999558)

[Примеры работы программы 6](#_Toc160999559)

[Вывод 9](#_Toc160999560)

# Описание метода

Метод Ньютона для решения СНУ (систем нелинейных уравнений) – это обычный метод Ньютона для решения уравнений, но работающий с матрицей и векторами. Метод основан на использовании разложения функции в ряд Тейлора окрестности текущего приближения. Метод Ньютона представляет собой итерационный процесс, который осуществляется по определенной формуле следующего вида:

Где W – матрица Якоби, то есть матрица частных производных функций системы по каждой неизвестной.

Критерием окончания итерационного процесса является неравенство:

Метод Ньютона крайне эффективен, быстр и точен. В дальнейших разделах отчёта можно увидеть реализацию данного метода.

# Блок-схема

Изображение выглядит как диаграмма, текст, План, Технический чертеж

Автоматически созданное описание

# Исходный код метода на языке программирования Python

|  |
| --- |
| def first\_deriative\_1(args: []) -> float:  return math.cos(args[0])   def first\_deriative\_2(args: []) -> float:  return 1   def second\_deriative\_1(args: []) -> float:  return args[1] / 2   def second\_deriative\_2(args: []) -> float:  return args[0] / 2   def third\_deriative\_1(args: []) -> float:  return (args[1] + k) / pow(math.cos(args[0] \* args[1] + k), 2) - 2 \* args[0]   def third\_deriative\_2(args: []) -> float:  return (args[0] + k) / pow(math.cos(args[0] \* args[1] + k), 2)   def fourth\_deriative\_1(args: []) -> float:  return 2 \* a \* args[0]   def fourth\_deriative\_2(args: []) -> float:  return 4 \* args[1]   def fifth\_deriative\_1(args: []) -> float:  return 2 \* args[0]   def fifth\_deriative\_2(args: []) -> float:  return 2 \* args[1]   def fifth\_deriative\_3(args: []) -> float:  return 2 \* args[2]   def six\_deriative\_1(args: []) -> float:  return 4 \* args[0]   def six\_deriative\_2(args: []) -> float:  return 2 \* args[1]   def six\_deriative\_3(args: []) -> float:  return -4   def seven\_deriative\_1(args: []) -> float:  return 6 \* args[0]   def seven\_deriative\_2(args: []) -> float:  return -4   def seven\_deriative\_3(args: []) -> float:  return 2 \* args[2]   def jacobian(n: int, args: []):  if n == 1:  return [[first\_deriative\_1(args), first\_deriative\_2(args)],  [second\_deriative\_1(args), second\_deriative\_2(args)]]  elif n == 2 or n == 3:  return [[third\_deriative\_1(args), third\_deriative\_2(args)],  [fourth\_deriative\_1(args), fourth\_deriative\_2(args)]]  elif n == 4:  return [[fifth\_deriative\_1(args), fifth\_deriative\_2(args), fifth\_deriative\_3(args)],  [six\_deriative\_1(args), six\_deriative\_2(args), six\_deriative\_3(args)],  [seven\_deriative\_1(args), seven\_deriative\_2(args), seven\_deriative\_3(args)]]  else:  return [[default\_function]]   def solve\_by\_fixed\_point\_iterations(system\_id, number\_of\_unknowns, initial\_approximations):  if system\_id > 4:  return initial\_approximations  eps = 1e-5  max\_iter = 1000  values = list(initial\_approximations)  funcs = get\_functions(system\_id)  it = 0  for \_ in range(max\_iter):  try:  f\_vals = [funcs[i](values) for i in range(number\_of\_unknowns)]  except ValueError:  return values  J = jacobian(system\_id, values)  J\_inv = [[J[j][i] for j in range(len(J))] for i in range(len(J[0]))]  delta = []  for i in range(len(J)):  s = sum(J\_inv[i][k] \* f\_vals[k] for k in range(len(f\_vals)))  delta.append(-s)  if all(abs(d) < eps for d in delta):  return values  for i in range(len(delta)):  values[i] += delta[i]  return values |

# Примеры работы программы

1. Первая система:

Изображение выглядит как текст, Шрифт, снимок экрана, дизайн

Автоматически созданное описание

1. Вторая система:

Изображение выглядит как текст, Шрифт, снимок экрана, дизайн

Автоматически созданное описание

1. Вторая система, но с другими параметрами:

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

1. Третья система:

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

1. Ввод system\_id больше, чем существует систем:

Изображение выглядит как снимок экрана

Автоматически созданное описание

Просто выходим из программы с приближёнными значениями.

# Вывод

При выполнении данной лабораторной работы я поработал с методом Ньютона для решения СНУ (систем нелинейных уравнений). Попробовав его на нескольких системах, я ощутил его недостатки и преимущества.

Основным преимуществом метода Ньютона является его быстрая сходимость в окрестности точного решения. Если начальное приближение достаточно близко к решению, то метод Ньютона сходится квадратично.

Однако метод Ньютона имеет ряд недостатков. Например, вычисление матрицы Якоби и её обратной матрицы может быть крайне сложным, особенно для систем с большим количеством неизвестных. С этой проблемой помогает справиться модифицированный метод Ньютона, в котором матрицу Якоби вычисляют всего один раз с приближёнными значениями. Но эта модификация проигрывает в точности и в скорости сходимости оригинальному методу. Также метод Ньютона может сходиться очень медленно или вообще может не сходиться, если приближённые значения далеки от решения.

Метод Ньютона имеют быструю сходимость, чем метод хорд или метод простых итераций, но он при этом требует гораздо большего количества вычислений на каждом проходе цикла.

Так как вычисление обратной матрицы требует O(N^3), то логично предположить, что весь метод выполняется за O(N^3). А если быть точнее, то за O(iterations \* N^3).

Оценка ошибки метода Ньютона зависит от степени гладкости функции и начального приближения.

В заключение метод Ньютона является эффективным методом решения СНУ, особенно когда начальное приближение находится близко к точному. Но метод Ньютона требует выполнить много вычислений для нахождения ответа.