**Московский авиационный институт**

**(национальный исследовательский университет)**

Институт №8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

**Курсовая работа по**

**численным методам**

**«Решение нелинейных интегральных уравнений»**

Студент: Сухотин Игорь

Группа: М8O-402Б

Дата: 25.02.21

*Москва*

*2021 г*

Оглавление

[Нелинейное уравнение Вольтерра II рода 2](#_Toc50906248)

[Метод квадратур 2](#_Toc50906249)

[Описание методы 2](#_Toc50906250)

[Пример 7](#_Toc50906251)

[Итерационные методы 12](#_Toc50906252)

[Описание методов 12](#_Toc50906253)

[Пример 14](#_Toc50906254)

[Нелинейное уравнение Фредгольма II рода 17](#_Toc50906255)

[Метод квадратур 17](#_Toc50906256)

[Описание методов 17](#_Toc50906257)

[Пример 23](#_Toc50906258)

[Итерационные методы 26](#_Toc50906259)

[Описание методов 26](#_Toc50906260)

[Пример 27](#_Toc50906261)

[Полный листинг программы 30](#_Toc50906262)

[Выводы 38](#_Toc50906263)

[Список литературы 39](#_Toc50906264)

# Нелинейное уравнение Вольтерра II рода

## Метод квадратур

### Описание методы

Нелинейное уравнение Вольтерра II рода имеет следующий вид:

В уравнении роль неизвестной играет функция на отрезке . Отмечу, что ядро интегрального уравнения зависит от искомой функции . Предположим, что ядро и правая часть этого уравнения таковы, что его решение существует и единственно в классе непрерывных на отрезке функций. Построим метод приближённого решения , основанный на аппроксимации интеграла с помощью квадратурной формулы.

Пусть на отрезке задана сетка с узлами Зафиксируем в уравнении значения переменной в узлах сетки. Получим следующие равенства:

Теперь заменим интегралы в этих равенствах конечными суммами. Получим систему нелинейных соотношений:

Здесь – веса квадратурной формулы при аппроксимации интеграла. Изучим их попристальнее.

Пусть необходимо вычислить следующий интеграл:

Не вдаваясь в подробности, для этой задачи можно использовать следующие формулы для равномерной сетки:

Если положить , , то аппроксимирующие формулы запишутся так:

Вектора – это и есть различные типы весов в формуле . Их инициализация в программе выглядит так:

*# инициализация массива коэффициентов перед ядром*

**if** method\_int == "rectangle":

A = np.concatenate(([0], h \* np.ones(n - 2, dtype=np.double), [0]))

**elif** method\_int == "trapeze":

A = np.concatenate(([h / 2], h \* np.ones(n - 2, dtype=np.double), [h / 2]))

**elif** method\_int == "simpson":

A = np.concatenate(([h / 3],

[4 \* h / 3, 2 \* h / 3] \* (n - 2 >> 1),

[4 \* h / 3] **if** n & 1 == 1 **else** [],

[h / 3]))

Вернёмся к соотношениям :

Если нам удастся решить эту систему нелинейных уравнений относительно , то уравнение Вольтерра будет решено.

Немного перепишем следующим образом:

и распишем систему более подробно:

Откуда легко видно, что мы имеем дело с рекуррентной системой нелинейных уравнений одной переменной. На каждом из шагов необходимо решить одно нелинейное уравнение относительно .

Для такой задачи хорошо подходит метод итераций, а также метод Ньютона. Их программное воплощение выглядит довольно просто:

**def** eq\_solve(f, x0, method="iter", der=None, abstol = 1e-10):

"""

Решает трансцендентное уравнение f(x)=0 или x=f(x)

Параметры:

f - функция

x0 - начальное приближение

method =

"iter" - решение x=f(x) методом итераций

"newt" - решение f(x)=0 метод Ньютона

der - частная производная f(x) для method="iter"

abstol - абсолютная погрешность вычислений

Возвращает:

x - решение трансцендентного уравнения f(x)=0 или x=f(x)

"""

x = x0

x\_ = np.Inf

**if** method == "iter": *# Метод итераций*

**while** np.abs(x - x\_) >= abstol:

x\_ = x

x = f(x)

**elif** method == "newt": *# метод Ньютона*

**while** np.abs(x - x\_) >= abstol:

x\_ = x

x = x - f(x) / der(x)

**return** x

Однако есть несколько небольших проблем:

1. Откуда взять начальное приближение ?
2. Как переписать уравнение в виде для метода итераций?
3. Откуда взять производную для метода Ньютона?
4. Сходимость методов?

Я решил эти вопросы так:

1. Для метода простых итераций:

т.е. нам нужно решить это уравнение относительно

1. Для метода Ньютона имеем:

т.е. для метода Ньютона необходимо задать частную производная ядра по искомой функции.

1. Метод простых итераций сходится в единственному решению на отрезке, когда

Проблема в том, что мы заранее не знаем , однако малость коэффициента (и чем меньше шаг сетки, тем он меньше) позволяет строить предположения о том, что итерационный процесс будет (не всегда) сходиться.

Для метода Ньютона необходимо, чтобы

и .

Учитывая эти соображения, получим следующий кусок программы:

**if** method == "quad": *# квадратурные методы*

y[0] = f(x[0])

**if** kind == "simple": *# решением нелинейного уравнения методом итераций*

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

y[i] = eq\_solve(**lambda** yi: A[i] \* K(x[i], x[i], yi) + f(x[i]) + np.dot(A[:i], K(x[i], x[:i], y[:i])),

x0 = f(x[i]), method="iter",

abstol=abstol)

**elif** kind == "newt": *# решением нелинейного уравнения методом Ньютона*

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

y[i] = eq\_solve(**lambda** yi: yi - A[i] \* K(x[i], x[i], yi) - f(x[i]) - np.dot(A[:i], K(x[i], x[:i], y[:i])),

x0 = f(x[i]), method="newt",

der=**lambda** yi: 1 - A[i] \* der(x[i], x[i], yi),

abstol=abstol)

### Пример

Будем численно решать следующее интегральное уравнение Вольтерра:

его точное решение есть .

Параметры:

Ввод уравнения и параметров в программу выглядит так:

*#%% Гиперпараметры*

h = 1e-1 *# шаг сетки*

abstol = 1e-10 *# абсолютная погрешность*

*#%% Параметры для уравнения (7) Вольтерра 2 рода*

yt = np.vectorize(**lambda** x: 1) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: np.exp(-(x - s)) \* y \*\* 2) *# ядро*

der = np.vectorize(**lambda** x, s, y: 2 \* np.exp(-(x - s)) \* y) *# ЧАСТНАЯ производная от ядра по y*

f = np.vectorize(**lambda** x: np.exp(-x)) *# правая часть*

xlim = (0, 1) *# отрезок интегрирования*

А типичный вызов программы так:

*#%% rectange + quad\_simple для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

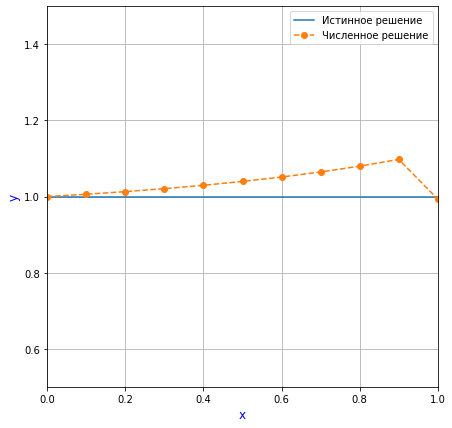
h=h,

method\_int="rectangle", method\_solve="quad\_simple",

abstol=abstol)

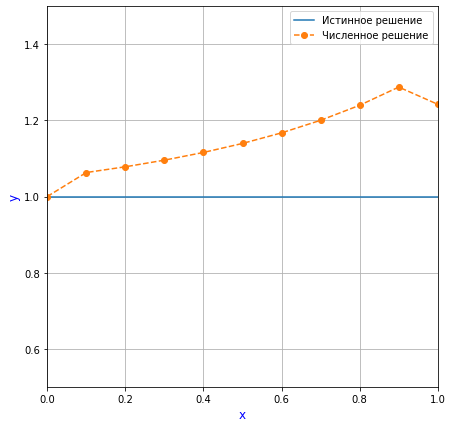
*Формула прямоугольников и метод итераций*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1. | 1 | 0. |
| 0.1 | 1.00605135 | 1 | 0.00605135 |
| 0.2 | 1.01291197 | 1 | 0.01291197 |
| 0.3 | 1.0207044 | 1 | 0.0207044 |
| 0.4 | 1.02957374 | 1 | 0.02957374 |
| 0.5 | 1.039693 | 1 | 0.039693 |
| 0.6 | 1.05126999 | 1 | 0.05126999 |
| 0.7 | 1.06455647 | 1 | 0.06455647 |
| 0.8 | 1.07986037 | 1 | 0.07986037 |
| 0.9 | 1.09756239 | 1 | 0.09756239 |
| 1.0 | 0.99311552 | 1 | 0.00688448 |



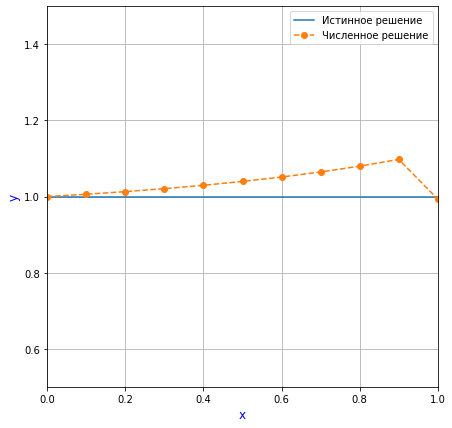
*Формула трапеций и метод итераций*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1. | 1 | 0. |
| 0.1 | 1.06309676 | 1 | 0.06309676 |
| 0.2 | 1.0781761 | 1 | 0.0781761 |
| 0.3 | 1.09561026 | 1 | 0.09561026 |
| 0.4 | 1.11586452 | 1 | 0.11586452 |
| 0.5 | 1.13952849 | 1 | 0.13952849 |
| 0.6 | 1.16736125 | 1 | 0.16736125 |
| 0.7 | 1.20035809 | 1 | 0.20035809 |
| 0.8 | 1.23985229 | 1 | 0.23985229 |
| 0.9 | 1.28767558 | 1 | 0.28767558 |
| 1.0 | 1.24230287 | 1 | 0.24230287 |



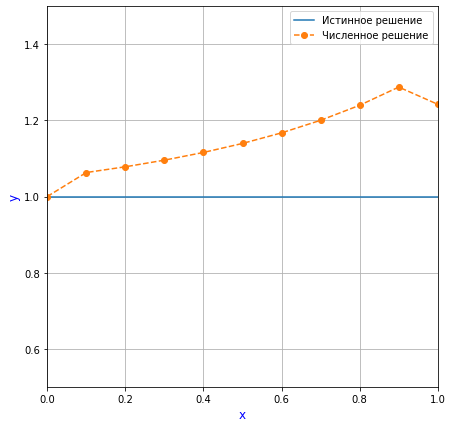
*Формула прямоугольников и метод Ньютона*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1. | 1 | 0. |
| 0.1 | 1.00605135 | 1 | 0.00605135 |
| 0.2 | 1.01291197 | 1 | 0.01291197 |
| 0.3 | 1.0207044 | 1 | 0.0207044 |
| 0.4 | 1.02957374 | 1 | 0.02957374 |
| 0.5 | 1.039693 | 1 | 0.039693 |
| 0.6 | 1.05126999 | 1 | 0.05126999 |
| 0.7 | 1.06455647 | 1 | 0.06455647 |
| 0.8 | 1.07986037 | 1 | 0.07986037 |
| 0.9 | 1.09756239 | 1 | 0.09756239 |
| 1.0 | 0.99311552 | 1 | 0.00688448 |



*Формула трапеций и метод Ньютона*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1. | 1 | 0. |
| 0.1 | 1.06309676 | 1 | 0.06309676 |
| 0.2 | 1.0781761 | 1 | 0.0781761 |
| 0.3 | 1.09561026 | 1 | 0.09561026 |
| 0.4 | 1.11586452 | 1 | 0.11586452 |
| 0.5 | 1.13952849 | 1 | 0.13952849 |
| 0.6 | 1.16736125 | 1 | 0.16736125 |
| 0.7 | 1.20035809 | 1 | 0.20035809 |
| 0.8 | 1.23985229 | 1 | 0.23985229 |
| 0.9 | 1.28767558 | 1 | 0.28767558 |
| 1.0 | 1.24230287 | 1 | 0.24230287 |



Замечаем, что чем ближе мы приближаемся к правому концу интервала, тем сильнее ошибка. Это связано с тем, что при последовательном решении (от левого края к правому) ошибка накапливается.

*Замечание*: я специально выбрал такую широкую сетку, чтобы эта тенденция была видна сильнее: чем меньше шаг сетки, тем лучше идёт аппроксимация.

## Итерационные методы

### Описание методов

Запишем нелинейное уравнение Вольтерра II рода в удобном для применения простой итерации виде:

Построим последовательность функций с помощью рекуррентного соотношения

В отличие от линейного случая данного уравнения, т.е. когда

нельзя однозначно сказать, при каком условии последовательность сойдётся к решению при нелинейном ядре . Напомню, что, пользуясь простейшими неравенствами для интегралов, можно вывести, когда будет сходиться последовательность при , а именно при:

где

но поскольку это условие является необходимым, но не достаточным, то последовательность приближений всё равно можно построить в надежде на лучшее.

Аппроксимируем интеграл в суммой и получим:

За начальное приближение можно принять . Итерационный процесс следует прекратить, когда

Замечу, что при вычислении на каждой итерации можно сразу обновлять, а не в конце итерации, вновь подсчитанный и тогда получится метод Зейделя.

Программная реализация этой идеи:

**elif** method == "iter": *# итерационные методы*

y = f(x) *# начальное приближение*

y\_ = np.Inf \* np.ones\_like(y)

**while** np.linalg.norm(y - y\_, ord=ord) >= abstol:

y\_ = np.copy(y) *# сохраняем старый массив*

y[0] = f(x[0])

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

y[i] = f(x[i]) + np.dot(A[:i + 1], K(x[i], x[:i + 1], y\_[:i + 1] **if** kind == "iter" **else** y[:i + 1]))

### Пример

Будем численно решать следующее интегральное уравнение Вольтерра:

его точное решение есть .

Параметры:

Ввод уравнения и параметров в программу выглядит так:

*#%% Гиперпараметры*

h = 2e-1 *# шаг сетки*

abstol = 1e-10 *# абсолютная погрешность*

*#%% Параметры для уравнения (10) Вольтерра 2 рода*

yt = np.vectorize(**lambda** x: x) *# истинное решение*

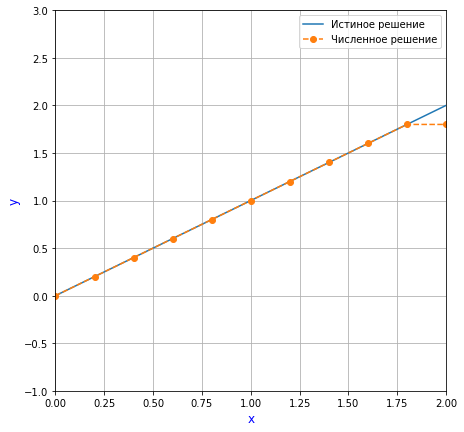
K = np.vectorize(**lambda** x, s, y: (1 + y \*\* 2) / (1 + s \*\* 2)) *# ядро y*

f = np.vectorize(**lambda** x: 0.) *# правая часть*

xlim = (0, 2) *# отрезок интегрирования*

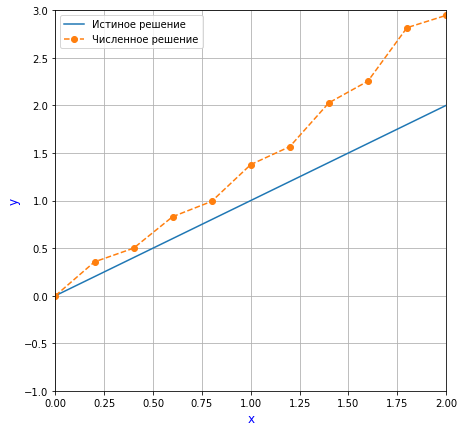
*Формула прямоугольников и метод Зейделя*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка[[1]](#footnote-1)** |
| 0.0 | 0.0 | 0.0 | 0.0 |
| 0.2 | 0.2 | 0.2 | 0.0 |
| 0.4 | 0.4 | 0.4 | 0.0 |
| 0.6 | 0.6 | 0.6 | 0.0 |
| 0.8 | 0.8 | 0.8 | 0.0 |
| 1.0 | 1.0 | 1.0 | 0.0 |
| 1.2 | 1.2 | 1.2 | 0.0 |
| 1.4 | 1.4 | 1.4 | 0.0 |
| 1.6 | 1.6 | 1.6 | 0.0 |
| 1.8 | 1.8 | 1.8 | 0.0 |
| 2.0 | 1.8 | 2.0 | 2e-01 |



*Формула Симпсона и метод простых итераций*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 0. | 0.0 | 0. |
| 0.2 | 0.35547813 | 0.2 | 0.15547813 |
| 0.4 | 0.49904683 | 0.4 | 0.09904683 |
| 0.6 | 0.83030202 | 0.6 | 0.23030202 |
| 0.8 | 0.99153267 | 0.8 | 0.19153267 |
| 1.0 | 1.3780799 | 1.0 | 0.3780799 |
| 1.2 | 1.56688469 | 1.2 | 0.36688469 |
| 1.4 | 2.02720566 | 1.4 | 0.62720566 |
| 1.6 | 2.25513142 | 1.6 | 0.65513142 |
| 1.8 | 2.81717325 | 1.8 | 1.01717325 |
| 2.0 | 2.94624469 | 2.0 | 0.94624469 |



# Нелинейное уравнение Фредгольма II рода

## Метод квадратур

### Описание методов

Нелинейное уравнение Фредгольма II рода в общем случае имеет вид

, как и в случае – неизвестная функция на отрезке . Не вдаваясь в вопросы существования решения решим его методом квадратур, а именно: введём сетку с постоянным шагом и аппроксимируем интеграл взвешенной суммой, как мы это делали в :

уже нельзя решить, как систему рекуррентных нелинейных уравнений одной переменной. Попробуем решить её полную систему нелинейных уравнений, относительно неизвестного вектора . Т.е.

Систему можно решать по-разному. Рассмотрим сначала итерационные методы.

Немного перепишем , чтобы применить метод простых итераций или метод Зейделя:

или

Теперь можно взять в качестве начального приближения и организовать итерационный процесс таким образом:

Также можно попробовать решить метод Ньютона. Кратко напомню его суть: пусть мы имеем систему нелинейных уравнений:

Процесс нахождения решения такой системы можно представить в виде:

где значения приращений определяются определяются из решения системы *линейных* алгебраических уравнений, которые получаются из формулы Тейлора для функции многих переменных:

Решить эту линейную систему в общем случае можно, например, методом *LUP*-разложения главной матрицы системы с последующим решением двух систем с треугольными матрицами методом исключения Гаусса.

Программная реализация этого метода:

**def** LUP(A):

"""

Вычисляет LUP-разложение для матрицы

Параметры:

A - входная матрица

Возвращает:

LU - матрица, такая, что ненулевые элементы L располагаются ниже главной диагонали LU,

а ненулевые элементы U - не ниже главной диагонали LU

pi - вектор перестановок

"""

LU = np.copy(A)

n = LU.shape[0]

pi = np.arange(n)

**for** k **in** range(n):

k\_ = np.argmax(np.abs(LU[k:, k])) + k

**if** k != k\_:

pi[k], pi[k\_] = pi[k\_], pi[k]

LU[k, :], LU[k\_, :] = LU[k\_, :], LU[k, :]

**for** i **in** range(k + 1, n):

LU[i, k] /= LU[k, k]

**for** j **in** range(k + 1, n):

LU[i, j] -= LU[i, k] \* LU[k, j]

**return** LU, pi

**def** LUP\_solve(LU, pi, b):

"""

Решает линейную систему Ax=b через LUP разложение матрицы A

Параметры:

LU - матрица, такая, что ненулевые элементы L располагаются ниже главной диагонали LU,

а ненулевые элементы U - не ниже главной диагонали LU

pi - вектор перестановок

b - правая часть

Возвращает:

x - решение системы Ax=b

"""

n = LU.shape[0]

x = np.zeros(n)

y = np.zeros(n)

**for** i **in** range(n):

y[i] = b[pi[i]] - np.dot(LU[i, :i], y[:i]) *# прямой ход*

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

x[i] = (y[i] - np.dot(LU[i, i + 1:], x[i + 1:])) / LU[i, i] *# обратный ход*

**return** x

Главная матрица линейной системы это просто матрица Якоби отображения :

а правая часть – это

т.е.

Программная реализация метода Ньютона и итерационных методов для решения системы:

**def** sys\_solve(F, x0, method="iter", Jac=None, ord=None, abstol = 1e-10):

"""

Решает систему трансцендентных уравнений F(x)=0 или x=F(x), где

F = [F1, F2, ..., Fn]

Параметры:

F - функция n переменных

x0 - начальное приближение

method =

"iter" - решение x=F(x) методом итераций

"zelder" - решение x=F(x) методом Зейделя

"newt" - решение F(x)=0 метод Ньютона (требуется Jac)

Jac - Якобиан отображения F

ord - вид нормы

abstol - абсолютная погрешность вычислений

Возвращает:

x - решение трансцендентного уравнения f(x)=0 или x=f(x)

"""

x = x0

x\_ = np.Inf \* np.ones\_like(x)

**if** method == "iter": *# Метод итераций*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

x = F(x)

**elif** method == "zelder": *# Метод Зейделя*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

**for** i **in** range(len(x)):

x[i] = F(x)[i]

**elif** method == "newt": *# метод Ньютона*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

A = Jac(x)

b = -F(x)

LU, pi = LUP(A)

x += LUP\_solve(LU, pi, b)

**return** x

В частном случае для нелинейной системы матрица Якоби имеет вид, :

Т.е. достаточно знать только частную производную ядра . За начальное приближение по методу Ньютона можно положить .

Программная реализация решения нелинейного уравнения Фредгольма II квадратурными методами:

*# решение, полученное разными методами*

**if** method == "quad": *# квадратурные методы*

**if** kind == "simple" **or** kind == "zelder": *# простые итерации или Зейдель*

*# инициализация вектор-функции x = F(x)*

F = **lambda** yi: np.array([l \* np.dot(A, K(x[i], x, yi)) + f(x[i]) **for** i **in** range(n)])

*# решение нелинейной системы уравнений итерационными методами*

y = sys\_solve(F, x0 = f(x),

method=kind,

ord=ord,

abstol=abstol)

**elif** kind == "newt": *# Ньютон*

*# инициализация вектор-функции F(x) = 0*

F = **lambda** yi: np.array([yi[i] - l \* np.dot(A, K(x[i], x, yi)) - f(x[i]) **for** i **in** range(n)])

*# инициализация Якобиана F(x)*

Jac = **lambda** yi: np.array([[1 - l \* A[i] \* der(x[i], x[i], yi[i]) **if** i == j **else**

- l \* A[j] \* der(x[i], x[j], yi[j])

**for** j **in** range(n)]

**for** i **in** range(n)])

*# решение нелинейной системы уравнений методом Ньютона*

y = sys\_solve(F, x0 = f(x),

method="newt",

ord=ord,

Jac=Jac,

abstol=abstol)

### Пример

Будем численно решать следующее нелинейное уравнение Фредгольма с помощью квадратурных методов:

Его точное решение есть , .

Ввод уравнения:

*#%% Гиперпараметры*

h = 1e-1 *# шаг сетки*

abstol = 1e-10 *# абсолютная погрешность*

*#%% Параметры для уравнения (1) Фредгольма 2 рода*

yt = np.vectorize(**lambda** x: np.exp(-(x \*\* 2 / 2))) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: x \* s \* y \*\* 2) *# ядро*

der = np.vectorize(**lambda** x, s, y: 2 \* x \* s \* y) *# ЧАСТНАЯ производная от ядра по y*

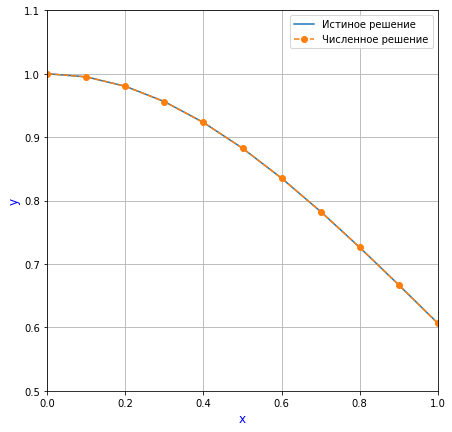
l = 0.125 *# лямбда*

f = np.vectorize(**lambda** x: np.exp(-(x \*\* 2 / 2)) + x \* (1 / np.e - 1) / 16) *# правая часть*

xlim = (0, 1) *# отрезок интегрирования*

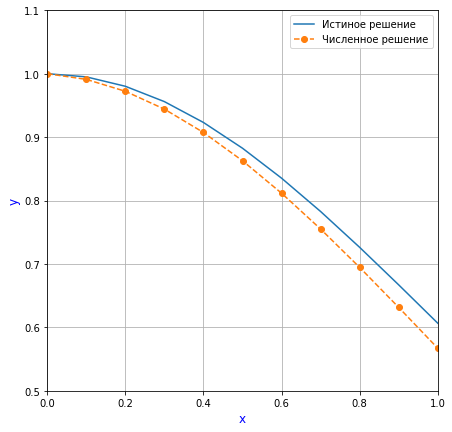
Инициализация весов по формуле Симпсона и решение нелинейной системы методом Ньютона:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1.00000000e+00 | 1.00000000e+00 | 0.00000000e+00 |
| 0.2 | 9.95007364e-01 | 9.95012479e-01 | 5.11557293e-06 |
| 0.4 | 9.80188442e-01 | 9.80198673e-01 | 1.02311459e-05 |
| 0.6 | 9.55982135e-01 | 9.55997482e-01 | 1.53467188e-05 |
| 0.8 | 9.23095884e-01 | 9.23116346e-01 | 2.04622917e-05 |
| 1.0 | 8.82471325e-01 | 8.82496903e-01 | 2.55778647e-05 |
| 1.2 | 8.35239518e-01 | 8.35270211e-01 | 3.06934376e-05 |
| 1.4 | 7.82668729e-01 | 7.82704538e-01 | 3.58090105e-05 |
| 1.6 | 7.26108112e-01 | 7.26149037e-01 | 4.09245835e-05 |
| 1.8 | 6.66930771e-01 | 6.66976811e-01 | 4.60401564e-05 |
| 2.0 | 6.06479504e-01 | 6.06530660e-01 | 5.11557293e-05 |



Инициализация весов по формуле Симпсона и решение нелинейной системы методом простых итераций:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **ошибка** |
| 0.0 | 1. | 1.00000000e+00 | 0. |
| 0.2 | 0.99106173 | 9.95012479e-01 | 0.00395075 |
| 0.4 | 0.97229717 | 9.80198673e-01 | 0.00790151 |
| 0.6 | 0.94414522 | 9.55997482e-01 | 0.01185226 |
| 0.8 | 0.90731333 | 9.23116346e-01 | 0.01580301 |
| 1.0 | 0.86274314 | 8.82496903e-01 | 0.01975377 |
| 1.2 | 0.81156569 | 8.35270211e-01 | 0.02370452 |
| 1.4 | 0.75504926 | 7.82704538e-01 | 0.02765527 |
| 1.6 | 0.69454301 | 7.26149037e-01 | 0.03160603 |
| 1.8 | 0.63142003 | 6.66976811e-01 | 0.03555678 |
| 2.0 | 0.56702312 | 6.06530660e-01 | 0.03950753 |



## Итерационные методы

### Описание методов

Подобно уравнению Вольтерра II рода уравнение Фредгольма можно решать итерационными методами.

Запишем нелинейное уравнение Фредгольма II рода в удобном для применения простой итерации виде:

Построим последовательность функций с помощью рекуррентного соотношения

В отличие от линейного случая данного уравнения, т.е. когда

нельзя однозначно сказать, при каком условии последовательность сойдётся к решению при нелинейном ядре . Пользуясь простейшими неравенствами для интегралов, можно вывести, когда будет сходиться последовательность при , а именно при:

где

но поскольку это условие является необходимым, но не достаточным, то последовательность приближений всё равно можно построить в надежде на лучшее.

Аппроксимируем интеграл в суммой и получим:

За начальное приближение можно принять . Итерационный процесс следует прекратить, когда

Замечу, что при вычислении на каждой итерации можно сразу обновлять, а не в конце итерации, вновь подсчитанный и тогда получится метод Зейделя.

Программная реализация этой идеи:

**elif** method == "iter": *# итерационные методы*

y = f(x) *# начальное приближение*

y\_ = np.Inf \* np.ones\_like(y)

**while** np.linalg.norm(y - y\_, ord=ord) >= abstol:

y\_ = np.copy(y) *# сохраняем старый массив*

**for** i **in** range(n):

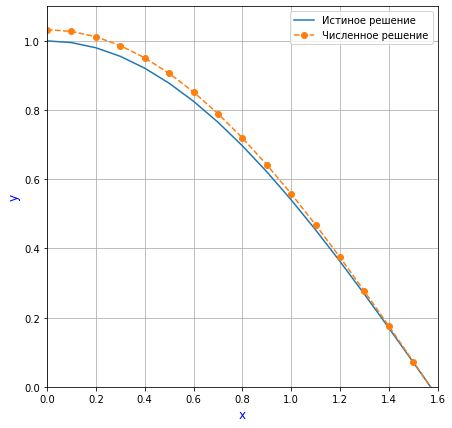
y[i] = f(x[i]) + l \* np.dot(A, K(x[i], x, y\_ **if** kind == "simple" **else** y))

### Пример

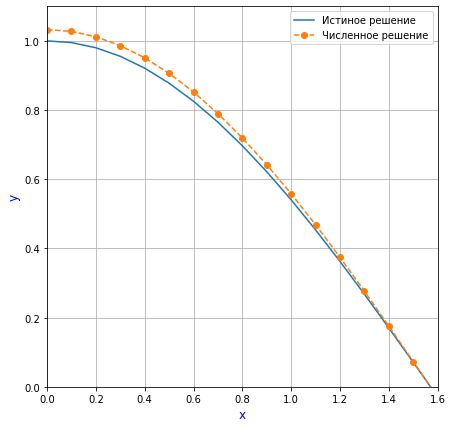
Будем численно решать следующее нелинейное уравнение Фредгольма с помощью итерационных методов:

Его точное решение есть , .

Веса по Симпсону и метод простых итераций:



Веса по формуле прямоугольников и метод Зейделя:



# Полный листинг программы

Все программы написаны в интерактивной среде *Spyder* на языке *Python*.

***num\_methods.py***

*# -\*- coding: utf-8 -\*-*

**import** numpy **as** np

**def** LUP(A):

"""

Вычисляет LUP-разложение для матрицы

Параметры:

A - входная матрица

Возвращает:

LU - матрица, такая, что ненулевые элементы L располагаются ниже главной диагонали LU,

а ненулевые элементы U - не ниже главной диагонали LU

pi - вектор перестановок

"""

LU = np.copy(A)

n = LU.shape[0]

pi = np.arange(n)

**for** k **in** range(n):

k\_ = np.argmax(np.abs(LU[k:, k])) + k

**if** k != k\_:

pi[k], pi[k\_] = pi[k\_], pi[k]

LU[k, :], LU[k\_, :] = LU[k\_, :], LU[k, :]

**for** i **in** range(k + 1, n):

LU[i, k] /= LU[k, k]

**for** j **in** range(k + 1, n):

LU[i, j] -= LU[i, k] \* LU[k, j]

**return** LU, pi

**def** LUP\_solve(LU, pi, b):

"""

Решает линейную систему Ax=b через LUP разложение матрицы A

Параметры:

LU - матрица, такая, что ненулевые элементы L располагаются ниже главной диагонали LU,

а ненулевые элементы U - не ниже главной диагонали LU

pi - вектор перестановок

b - правая часть

Возвращает:

x - решение системы Ax=b

"""

n = LU.shape[0]

x = np.zeros(n)

y = np.zeros(n)

**for** i **in** range(n):

y[i] = b[pi[i]] - np.dot(LU[i, :i], y[:i]) *# прямой ход*

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

x[i] = (y[i] - np.dot(LU[i, i + 1:], x[i + 1:])) / LU[i, i] *# обратный ход*

**return** x

**def** eq\_solve(f, x0, method="iter", der=None, abstol = 1e-10):

"""

Решает трансцендентное уравнение f(x)=0 или x=f(x)

Параметры:

f - функция

x0 - начальное приближение

method =

"iter" - решение x=f(x) методом итераций

"newt" - решение f(x)=0 метод Ньютона

der - частная производная f(x) для method="iter"

abstol - абсолютная погрешность вычислений

Возвращает:

x - решение трансцендентного уравнения f(x)=0 или x=f(x)

"""

x = x0

x\_ = np.Inf

**if** method == "iter": *# Метод итераций*

**while** np.abs(x - x\_) >= abstol:

x\_ = x

x = f(x)

**elif** method == "newt": *# метод Ньютона*

**while** np.abs(x - x\_) >= abstol:

x\_ = x

x = x - f(x) / der(x)

**return** x

*# с несколькими неизвестными корнями, т.е. систему*

**def** sys\_solve(F, x0, method="iter", Jac=None, ord=None, abstol = 1e-10):

"""

Решает систему трансцендентных уравнений F(x)=0 или x=F(x), где

F = [F1, F2, ..., Fn]

Параметры:

F - функция n переменных

x0 - начальное приближение

method =

"iter" - решение x=F(x) методом итераций

"zelder" - решение x=F(x) методом Зейделя

"newt" - решение F(x)=0 метод Ньютона (требуется Jac)

Jac - Якобиан отображения F

ord - вид нормы

abstol - абсолютная погрешность вычислений

Возвращает:

x - решение трансцендентного уравнения f(x)=0 или x=f(x)

"""

x = x0

x\_ = np.Inf \* np.ones\_like(x)

**if** method == "iter": *# Метод итераций*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

x = F(x)

**elif** method == "zelder": *# Метод Зейделя*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

**for** i **in** range(len(x)):

x[i] = F(x)[i]

**elif** method == "newt": *# метод Ньютона*

**while** np.linalg.norm(x - x\_, ord=ord) >=abstol:

x\_ = x

A = Jac(x)

b = -F(x)

LU, pi = LUP(A)

x += LUP\_solve(LU, pi, b)

**return** x

**def** volt\_II\_nonlin\_solve(K, f,

xlim,

h=1e-2,

method\_int="rectangle", method\_solve="iter",

der=None,

ord=None,

abstol = 1e-10):

"""

Решает нелинейное уравнение Вольтерра 2 рода

Параметры:

K - ядро: функция 3 переменных x, s, y

f - правая часть

xlim - отрезок интегрирования

h - шаг сетки

method\_int =

"rectangle" - инициализация массива коэффициентов перед ядром

по формуле прямоугольников

"trapeze" - ... по формуле трапеций

"simpson" - ... по формуле Симпсона (парабол)

method\_solve =

"quad\_simple" - решение методом квадратур, в процессе решается одномерное

трансцендентных уравнение методом простых итераций

"quad\_newt" - решение методом квадратур, в процессе решается одномерное

трансцендентных уравнение методом Ньютона

(требуется der)

"iter\_simple" - решение методом итераций

"iter\_zelder" - решение методом Зейделя

der - частная производная ядра по y

ord - вид нормы

abstol - абсолютная погрешность вычислений

Возвращает:

x - массив аргументов

y - массив значений

"""

x = np.arange(xlim[0], xlim[1] + h, h, dtype=np.double)

n = len(x) *# длина сетки*

y = np.zeros(shape=n, dtype=np.double)

*# инициализация массива коэффициентов перед ядром*

**if** method\_int == "rectangle":

A = np.concatenate(([0], h \* np.ones(n - 2, dtype=np.double), [0]))

**elif** method\_int == "trapeze":

A = np.concatenate(([h / 2], h \* np.ones(n - 2, dtype=np.double), [h / 2]))

**elif** method\_int == "simpson":

A = np.concatenate(([h / 3],

[4 \* h / 3, 2 \* h / 3] \* (n - 2 >> 1),

[4 \* h / 3] **if** n & 1 == 1 **else** [],

[h / 3]))

method, kind = method\_solve.split(sep="\_")

*# решение, полученное разными методами*

**if** method == "quad": *# квадратурные методы*

y[0] = f(x[0])

**if** kind == "simple": *# решением нелинейного уравнения методом итераций*

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

y[i] = eq\_solve(**lambda** yi: A[i] \* K(x[i], x[i], yi) + f(x[i]) + np.dot(A[:i], K(x[i], x[:i], y[:i])),

x0 = f(x[i]), method="iter",

abstol=abstol)

**elif** kind == "newt": *# решением нелинейного уравнения методом Ньютона*

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

y[i] = eq\_solve(**lambda** yi: yi - A[i] \* K(x[i], x[i], yi) - f(x[i]) - np.dot(A[:i], K(x[i], x[:i], y[:i])),

x0 = f(x[i]), method="newt",

der=**lambda** yi: 1 - A[i] \* der(x[i], x[i], yi),

abstol=abstol)

**elif** method == "iter": *# итерационные методы*

y = f(x) *# начальное приближение*

y\_ = np.Inf \* np.ones\_like(y)

**while** np.linalg.norm(y - y\_, ord=ord) >= abstol:

y\_ = np.copy(y) *# сохраняем старый массив*

y[0] = f(x[0])

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

y[i] = f(x[i]) + np.dot(A[:i + 1], K(x[i], x[:i + 1], y\_[:i + 1] **if** kind == "iter" **else** y[:i + 1]))

**return** x, y

*# Фредгольма 2 рода*

**def** fred\_II\_nonlin\_solve(K, f, l,

xlim,

h=1e-2,

method\_int="rectangle", method\_solve="quad\_newt",

der=None,

ord=None,

abstol = 1e-10):

"""

Решает нелинейное уравнение Фредгольма 2 рода

Параметры:

K - ядро: функция 3 переменных x, s, y

f - правая часть

l - lambda

xlim - отрезок интегрирования

h - шаг сетки

method\_int =

"rectangle" - инициализация массива коэффициентов перед ядром

по формуле прямоугольников

"trapeze" - ... по формуле трапеций

"simpson" - ... по формуле Симпсона (парабол)

method\_solve =

"quad\_simple" - решение методом квадратур, в процессе решается система

трансцендентных уравнение методом простых итераций

"quad\_zelder" - решение методом квадратур, в процессе решается система

трансцендентных уравнение методом Зейделя

"quad\_newt" - решение методом квадратур, в процессе решается система

трансцендентных уравнение методом Ньютона

(требуется der)

"iter\_simple" - решение методом итераций

"iter\_zelder" - решение методом Зейделя

der - частная производная ядра по y

ord - вид нормы

abstol - абсолютная погрешность вычислений

Возвращает:

x - массив аргументов

y - массив значений

"""

x = np.arange(xlim[0], xlim[1] + h, h, dtype=np.double)

n = len(x) *# длина сетки*

y = np.zeros\_like(x)

*# инициализация массива коэффициентов перед ядром*

**if** method\_int == "rectangle":

A = np.concatenate(([0], h \* np.ones(n - 2, dtype=np.double), [0]))

**elif** method\_int == "trapeze":

A = np.concatenate(([h / 2], h \* np.ones(n - 2, dtype=np.double), [h / 2]))

**elif** method\_int == "simpson":

A = np.concatenate(([h / 3],

[4 \* h / 3, 2 \* h / 3] \* (n - 2 >> 1),

[4 \* h / 3] **if** n & 1 == 1 **else** [],

[h / 3]))

method, kind = method\_solve.split(sep="\_")

*# решение, полученное разными методами*

**if** method == "quad": *# квадратурные методы*

**if** kind == "simple" **or** kind == "zelder": *# простые итерации или Зейдель*

*# инициализация вектор-функции x = F(x)*

F = **lambda** yi: np.array([l \* np.dot(A, K(x[i], x, yi)) + f(x[i]) **for** i **in** range(n)])

*# решение нелинейной системы уравнений итерационными методами*

y = sys\_solve(F, x0 = f(x),

method=kind,

ord=ord,

abstol=abstol)

**elif** kind == "newt": *# Ньютон*

*# инициализация вектор-функции F(x) = 0*

F = **lambda** yi: np.array([yi[i] - l \* np.dot(A, K(x[i], x, yi)) - f(x[i]) **for** i **in** range(n)])

*# инициализация Якобиана F(x)*

Jac = **lambda** yi: np.array([[1 - l \* A[i] \* der(x[i], x[i], yi[i]) **if** i == j **else**

- l \* A[j] \* der(x[i], x[j], yi[j])

**for** j **in** range(n)]

**for** i **in** range(n)])

*# решение нелинейной системы уравнений методом Ньютона*

y = sys\_solve(F, x0 = f(x),

method="newt",

ord=ord,

Jac=Jac,

abstol=abstol)

**elif** method == "iter": *# итерационные методы*

y = f(x) *# начальное приближение*

y\_ = np.Inf \* np.ones\_like(y)

**while** np.linalg.norm(y - y\_, ord=ord) >= abstol:

y\_ = np.copy(y) *# сохраняем старый массив*

**for** i **in** range(n):

y[i] = f(x[i]) + l \* np.dot(A, K(x[i], x, y\_ **if** kind == "simple" **else** y))

**return** x, y

***main.py***

*# -\*- coding: utf-8 -\*-*

*# Либы*

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**from** num\_methods **import** volt\_II\_nonlin\_solve, fred\_II\_nonlin\_solve

*#%% Гиперпараметры*

h = 1e-1 *# шаг сетки*

abstol = 1e-10 *# абсолютная погрешность*

*#%% Параметры для уравнения (1) Вольтерра 2 рода*

yt = np.vectorize(**lambda** x: 1) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: np.exp(-(x - s)) \* y \*\* 2) *# ядро*

der = np.vectorize(**lambda** x, s, y: 2 \* np.exp(-(x - s)) \* y) *# ЧАСТНАЯ производная от ядра по y*

f = np.vectorize(**lambda** x: np.exp(-x)) *# правая часть*

xlim = (0, 1) *# отрезок интегрирования*

*#%% Параметры для уравнения (2) Вольтерра 2 рода*

yt = np.vectorize(**lambda** x: x) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: (1 + y \*\* 2) / (1 + s \*\* 2)) *# ядро*

der = np.vectorize(**lambda** x, s, y: 2 \* y / (1 + s \*\* 2)) *# ЧАСТНАЯ производная от ядра по y*

f = np.vectorize(**lambda** x: 0.) *# правая часть*

xlim = (0, 2) *# отрезок интегрирования*

*#%% Параметры для уравнения (1) Фредгольма 2 рода*

yt = np.vectorize(**lambda** x: np.exp(-(x \*\* 2 / 2))) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: x \* s \* y \*\* 2) *# ядро*

der = np.vectorize(**lambda** x, s, y: 2 \* x \* s \* y) *# ЧАСТНАЯ производная от ядра по y*

l = 0.125 *# лямбда*

f = np.vectorize(**lambda** x: np.exp(-(x \*\* 2 / 2)) + x \* (1 / np.e - 1) / 16) *# правая часть*

xlim = (0, 1) *# отрезок интегрирования*

*#%% Параметры для уравнения (2) Фредгольма 2 рода*

yt = np.vectorize(**lambda** x: np.cos(x)) *# истинное решение*

K = np.vectorize(**lambda** x, s, y: np.cos(x) \* np.sin(s) \* np.log(1 + y)) *# ядро*

der = np.vectorize(**lambda** x, s, y: np.cos(x) \* np.sin(s) / (1 + y)) *# ЧАСТНАЯ производная от ядра по y*

l = 0.125 *# лямбда*

f = np.vectorize(**lambda** x: 0.125 \* np.cos(x) \* (10 - np.log(2) - 1 / np.log(2))) *# правая часть*

xlim = (0, np.pi / 2) *# отрезок интегрирования*

*#%% rectange + quad\_simple для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="rectangle", method\_solve="quad\_simple",

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% trapeze + quad\_simple для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="trapeze", method\_solve="quad\_simple",

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% rectangle + quad\_newt для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="rectangle", method\_solve="quad\_newt",

der=der,

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% trapeze + quad\_newt для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="trapeze", method\_solve="quad\_newt",

der=der,

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% rectangle + iter\_zelder для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="rectangle", method\_solve="iter\_zelder",

abstol=1e-10)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% trapeze + iter\_simple для Вольтерра*

x, y = volt\_II\_nonlin\_solve(K, f,

xlim,

h=h,

method\_int="simpson", method\_solve="iter\_simple",

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% simpson + quad\_newt для Фредгольма*

x, y = fred\_II\_nonlin\_solve(K, f, l,

xlim,

h=h,

method\_int="simpson", method\_solve="quad\_newt",

der=der,

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% simpson + quad\_simple для Фредгольма*

x, y = fred\_II\_nonlin\_solve(K, f, l,

xlim,

h=h,

method\_int="simpson", method\_solve="quad\_simple",

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% simpson + iter\_simple для Фредгольма*

x, y = fred\_II\_nonlin\_solve(K, f, l,

xlim,

h=h,

method\_int="simpson", method\_solve="iter\_simple",

abstol=abstol)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% rectangle + iter\_zelder для Фредгольма*

x, y = fred\_II\_nonlin\_solve(K, f, l,

xlim,

h=h,

method\_int="rectangle", method\_solve="iter\_zelder",

abstol=1e-10)

**print**("x**\t**y**\t**yt**\t**err")

**print**(np.concatenate((x.reshape((-1, 1)),

y.reshape((-1, 1)),

yt(x).reshape((-1, 1)),

np.abs(yt(x) - y).reshape((-1, 1))),

axis=1))

*#%% График для уравнения Вольтерра (1)*

plt.figure(figsize=(7, 7))

plt.plot(x, yt(x), "-", label="Истинное решение")

plt.plot(x, y, marker='o', linestyle='dashed', label="Численное решение")

plt.xlim((x[0], x[-1]))

plt.ylim((0.5, 1.5))

plt.xlabel('x', fontsize=12, color='blue')

plt.ylabel('y', fontsize=12, color='blue')

plt.legend()

plt.grid(True)

*#%% График для уравнения Вольтерра (2)*

plt.figure(figsize=(7, 7))

plt.plot(x, yt(x), "-", label="Истиное решение")

plt.plot(x, y, marker='o', linestyle='dashed', label="Численное решение")

plt.xlim((x[0], x[-1]))

plt.ylim((-1, 3))

plt.xlabel('x', fontsize=12, color='blue')

plt.ylabel('y', fontsize=12, color='blue')

plt.legend()

plt.grid(True)

*#%% График для уравнения Фредгольма (1)*

plt.figure(figsize=(7, 7))

plt.plot(x, yt(x), "-", label="Истиное решение")

plt.plot(x, y, marker='o', linestyle='dashed', label="Численное решение")

plt.xlim((x[0], x[-1]))

plt.ylim((0.5, 1.1))

plt.xlabel('x', fontsize=12, color='blue')

plt.ylabel('y', fontsize=12, color='blue')

plt.legend()

plt.grid(True)

*#%% График для уравнения Фредгольма (2)*

plt.figure(figsize=(7, 7))

plt.plot(x, yt(x), "-", label="Истиное решение")

plt.plot(x, y, marker='o', linestyle='dashed', label="Численное решение")

plt.xlim((x[0], x[-1]))

plt.ylim((0.0, 1.1))

plt.xlabel('x', fontsize=12, color='blue')

plt.ylabel('y', fontsize=12, color='blue')

plt.legend()

plt.grid(True)

# Выводы

В ходе курсовой работы продемонстрированы методы решения нелинейных интегральных уравнений Вольтерра и Фредгольма II рода. Для их решения применились методы квадратур, а также итерационные методы.

Получены практические навыки численного решения таких уравнений в среде *Spyder* на языке *Python*.

# Список литературы

**В.Я. Дерр** Функциональный анализ: лекции и упражнения [Книга]. - 2013.

**Е.М. Карчевский** Численные решения интегральных уравнений и комплекс программ на языке Matlab. Учебное пособие [Книга]. - 2019.

**Формалёв В.Ф. Ревизников Д.Л.** Численные методы [Книга]. - 2004.

1. на самом деле там есть ошибка, но она в пределах 1e-12 [↑](#footnote-ref-1)