**Методы численного анализа**

**Отчёт по лабораторной работе**

**“Численные методы интегральных уравнений”**

**Вариант 10**

**Студент**

Малиев Эмиль Енгибарович

**Преподаватель**

Будник Анатолий Михайлович

ФПМИ БГУ  
2022

**Постановка задачи**

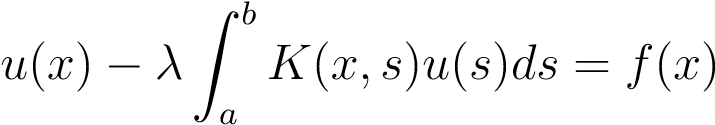
Даны интегральные уравнения Фредгольма и Вольтерра при следующих значениях их параметров:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |

**Алгоритм**

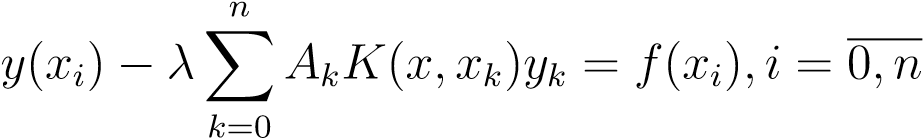
## **ИУФ-2**

Рассматриваем интегральное уравнение Фредгольма второго рода:



**Метод механических квадратур**

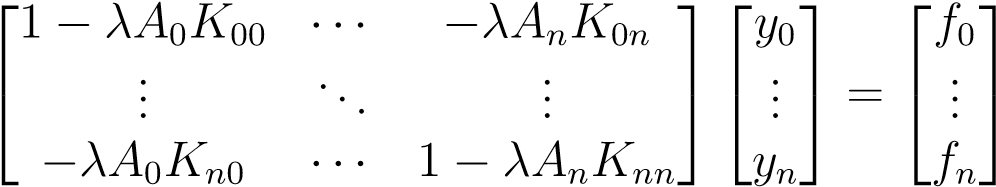
Вместо интеграла будем рассматривать квадратурную сумму:



Так как мы используем составную формулу правых прямоугольников, то коэффициенты квадратуры

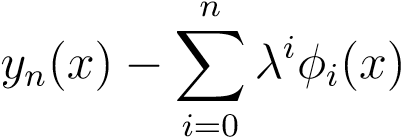
,  *k = 1,…,N*

Запишем (2) в матричном виде:

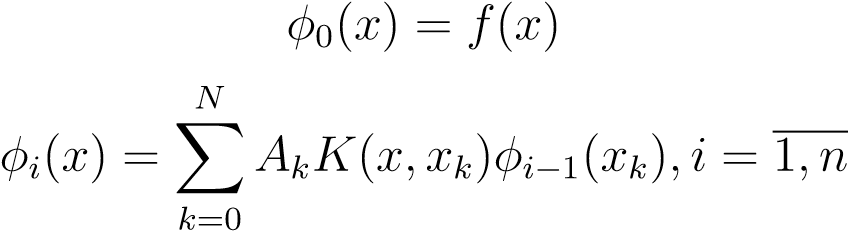


Чтобы узнать значение в произвольной точке применим следующую формулу:

|  |  |
| --- | --- |
| (1)  **Метод последовательных приближений Решение** будем искать в следующем виде: |  |

 (2)

*ϕi* находятся следующим образом:



Так как мы используем составную формулу правых прямоугольников, то коэффициенты квадратуры

,  *k = 1,…,N*

Чтобы узнать значение в произвольной точке применим формулу (1).

**ИУВ-2**

Рассматриваем интегральное уравнение Вольтерра второго рода:

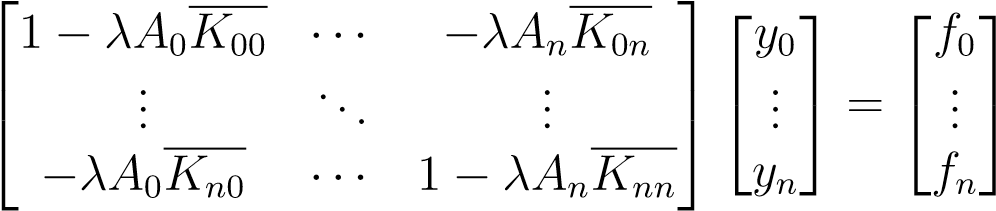
## 

**Метод механических квадратур**

Сведём уравнение (8) к уравнению Фредгольма с помощью следующей замены:

## 

Повторяя схему действий, проделанную при описании алгоритма для решения ИУФ-2, получаем следующую систему:



Чтобы узнать значение в произвольной точке применим формулу (1).

Используем составную формулу средних прямоугольников

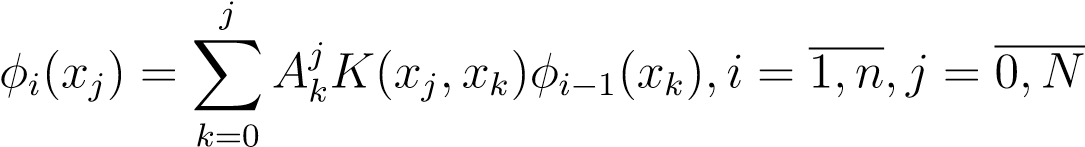
(3) ,

, *k =0, 1,…, N-1*

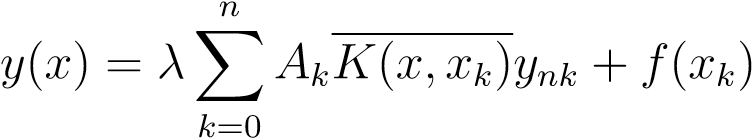
Метод последовательных приближений

Решение будем искать в виде (2). *ϕi* находятся следующим образом:

*ϕ*0(*x*) = *f*(*x*)



Коэффициенты квадратуры высчитываются по формуле (3). Чтобы узнать значение в произвольной точке применим следующую формулу:



**Листинг**

def k(x, s):

return np.exp(-x - s)

def k1(x, s):

return np.exp(-x - s) \* (s <= x)

def f(x):

return 0.5 \* (np.exp(-x) + np.exp(-3 \* x))

def method\_of\_mechanical\_quadratures(k, f, N=10, l=1., a=0., b=1.):

h = (b - a) / N

n = N + 1

coefficients = np.full(shape=n, fill\_value=h)

coefficients[0] = 0

x = np.arange(a, b + h, h)

A = np.zeros((n, n))

K = np.zeros((n, n))

for i in range(n):

for j in range(n):

K[i][j] = k(x[i], x[j])

for i in range(n):

for j in range(n):

A[i][j] = 1 - l \* coefficients[i] \* K[i][j] if i == j \

else -l \* h \* K[i][j]

F = np.zeros(n)

for i in range(n):

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

y = np.linalg.solve(A, F)

return y, coefficients

def value\_in\_point(value, k, f, N=10, l=1., a=0., b=1.):

h = (b - a) / N

n = N + 1

x = np.arange(a, b + h, h)

y, coefficients = \

method\_of\_mechanical\_quadratures(k, f, N, l, a, b)

summands = np.zeros(n)

for i in range(n):

summands[i] = coefficients[i] \* y[i] \* k(value, x[i])

return l \* sum(summands) + f(value)

print(value\_in\_point(0.5, k, f, l=0.3))

points = np.arange(0, 1., 0.1)

for i in range(10):

print(value\_in\_point(points[i], k, f, l=0.3))

Volterra MMK:

import numpy as np

def k(x, s):

return np.exp(-x - s)

def k1(x, s):

return np.exp(-x - s) \* (s <= x)

def f(x):

return 0.5 \* (np.exp(-x) + np.exp(-3 \* x))

def method\_of\_mechanical\_quadratures(k, f, N=100, l=0.1, a=0., b=1):

h = (b - a) / N

n = N + 1

coefficients = np.full(shape=n, fill\_value=h)

x = np.arange(a, b + h, h)

A = np.zeros((n, n))

K = np.zeros((n, n))

for i in range(n):

for j in range(n):

if i<=j:

K[i][j] = k(x[i], x[j] + h / 2)

else:

K[i][j] = 0

for i in range(n):

for j in range(n):

A[i][j] = 1 - l \* coefficients[i] \* K[i][j] if i == j \

else -l \* h \* K[i][j]

F = np.zeros(n)

for i in range(n):

F[i] = f(x[i] + h / 2)

y = np.linalg.solve(A, F)

return y, coefficients

def value\_in\_point(value, k, f, N=100, l=0.1, a=0., b=1.):

h = (b - a) / N

n = N + 1

x = np.arange(a, b + h, h)

y, coefficients = \

method\_of\_mechanical\_quadratures(k, f, N, l, a, b)

summands = np.zeros(n)

for i in range(n):

summands[i] = coefficients[i] \* y[i] \* k(value, x[i] + h / 2)

return l \* sum(summands) + f(value)

MMP:

def k(x, s):

return np.exp(-x - s)

def k1(x, s):

return np.exp(-x - s) \* (s <= x)

def f(x):

return 0.5 \* (np.exp(-x) + np.exp(-3 \* x))

def value\_in\_point(value, K, f, method, equ\_type, n=10, N=10, l=0.3, a=0., b=1.):

h = (b - a) / N

x = np.arange(a, b + h, h)

summands = np.zeros(N + 1)

y, coefficients = fixed\_point\_iteration(f, k, method, equ\_type, N, n, l, a, b)

for i in range(N + 1):

summands[i] = coefficients[i] \* y[i] \* K(value, x[i])

return l \* sum(summands) + f(value)

def fixed\_point\_iteration(f, k, method, equ\_type, N=10, n=10, l=0.3, a=0., b=1.):

h = (b - a) / N

x = np.array([a + h \* i for i in range(N + 1)])

coefficients = np.full(shape=N + 1, fill\_value=h)

coefficients[0] = 0

y = np.zeros(N + 1)

if method == 'Medium Riemann rule':

coefficients[0], coefficients[N] = h, h

for i in range(N - 1):

x[i] = x[i] + h/2

phi = np.zeros((n + 1, N + 1))

for i in range(n + 1):

for j in range(N + 1):

if i == 0:

phi[i][j] = f(x[j])

else:

phi[i][j] = np.sum([coefficients[m] \* k(x[j], x[m])

\* phi[i - 1][m] for m in range(N + 1)])

for i in range(n + 1):

for j in range(N + 1):

y[j] += l\*\*i \* phi[i][j]

return y, coefficients

def phi\_fredholm(i, x):

return (1 / (4 \* (2 \*\* i))) \* (np.exp(2) - 1) \*\* i \* (1 + 3 \* np.exp(2)) \* np.exp(-x - (2 + 2 \* i))

def solution\_fredholm(x, n, f, lmb):

sol = f(x)

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

sol += phi\_fredholm(i, x) \* (lmb \*\* i)

return sol

def phi\_volterra(i, x):

return (1 / m.factorial(i+1)) \* np.exp(-x \* (i+2)) \* (np.sinh(x) \*\* i) \* (i \* np.sinh(x) + (i+1) \* np.cosh(x))

def solution\_volterra(x, n, f, lmb):

sol = f(x)

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

sol += phi\_volterra(i, x) \* (lmb \*\* i)

return sol

x = np.arange(0., 1. + .1, .1)

print('Fredholm in nodes')

for i in range(len(x)):

print("x", i,

value\_in\_point(x[i], k, f, 'Right Riemann sum', 'Fredholm', l=0.3, N=10),

solution\_fredholm(x[i], 100, f, 0.3),

solution\_fredholm(x[i], 100, f, 0.3) - value\_in\_point(x[i], k, f, 'Right Riemann sum', 'Fredholm', l=0.3, N=10)

)

print('Fredholm in 1/2.2')

print (

value\_in\_point(1/2.2, k, f, 'Right Riemann sum', 'Fredholm', l=0.3, N=10),

solution\_fredholm(1/2.2, 100, f, 0.3),

solution\_fredholm(1/2.2, 100, f, 0.3) - value\_in\_point(1/2.2, k, f, 'Right Riemann sum', 'Fredholm', l=0.3, N=10)

)

print()

print('Volterra in nodes')

for i in range(len(x)):

print("x", i,

value\_in\_point(x[i], k1, f, 'Medium Riemann rule', 'Volterra', l=0.3, N=10),

solution\_volterra(x[i], 100, f, 0.3),

solution\_volterra(x[i], 100, f, 0.3) - value\_in\_point(x[i], k1, f, 'Medium Riemann rule', 'Volterra', l=0.3, N=10)

)

print('Volterra in 1/2.2')

print (

value\_in\_point(1/2.2, k1, f, 'Medium Riemann rule', 'Volterra', l=0.3, N=10),

solution\_volterra(1/2.2, 100, f, 0.3),

solution\_volterra(1/2.2, 100, f, 0.3) - value\_in\_point(1/2.2, k1, f, 'Medium Riemann rule', 'Volterra', l=0.3, N=10)

)

q = 0.3

print((0.3 \*\* 6) / 1.4)

print((q \*\* 6 / ((1 - (q / 6)) \* np.math.factorial(6))) \* 0.5)

points = np.arange(0, 1., 0.1)

**Результаты**

ММК для ИУФ-2: 0.5116685111581862

ММК для ИУВ-2: 0.49334945259279195

Нужные для подсчёта оценки на отрезке [0, 1]:

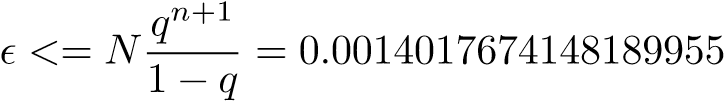
N = max *f*(*x*) = 0.5

M = max *K*(*x,s*) = 1

q = |*λ*|*M*(*b* − *a*) = 0.3

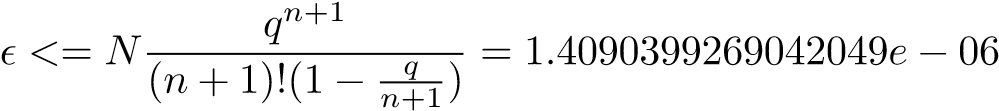
Погрешности метода последовательных приближений вычисляются по формулам:

ИУФ-2:



0.0005207142857142856

ИУВ-2:



Fredholm in nodes

x0 1.1002686084066393 1.1168137230065578 0.016545114599918476

x1 0.9135546080995611 0.9285252468752601 0.01497063877569893

x2 0.7658641878568533 0.7794101819930062 0.013545994136152872

x3 0.647974752281197 0.6602316746400837 0.012256922358886646

x4 0.5529691871769886 0.5640597091572709 0.011090521980282264

x5 0.4756463951358783 0.48568151440918794 0.010035119273309634

x6 0.4120838411863255 0.42116399259926984 0.009080151412944326

x7 0.3593127833885486 0.3675288441484128 0.008216060759864197

x8 0.31507704865214764 0.3225112478565298 0.0074341992043821525

x9 0.27765376024272514 0.28438050185598324 0.0067267416132580915

x10 0.24572001439732583 0.25180662191046127 0.006086607513135445

Fredholm in 1/2.2

0.508876926858847 0.5193787136509557 0.01050178679210867

Volterra in nodes

x0 1.0306037401104315 1.0 -0.030603740110431543

x1 0.8714573012145096 0.8466431215318873 -0.02481417968262234

x2 0.7422327113188888 0.7218841695632169 -0.02034854175567191

x3 0.6366663426623398 0.6197816479507641 -0.01688469471157572

x4 0.5498132428564103 0.53564025092299 -0.014172991933420365

x5 0.4777984156239576 0.46577272698755984 -0.012025688636397747

x6 0.41759850489320427 0.40729476536361436 -0.010303739529589906

x7 0.3668591607267767 0.35795433362508083 -0.008904827101695878

x8 0.32374684316147 0.31599313326125433 -0.007753709900215677

x9 0.28691519009312627 0.28003613826608226 -0.006879051827044014

x10 0.2551272245978691 0.24900469692705435 -0.006122527670814726

Volterra in 1/2.2

0.5058885143710122 0.495961866253996 -0.009926648117016212

Однако, следует отметить, что истинная погрешность больше, так как не учитывается то, что интегралы были посчитаны не точно, а с использованием квадратурных формул.

К недостаткам МПП можно отнести то, что на каждом шаге накапливается всё большая погрешность, так для подсчёта функции на i-ом шаге используется значение (i-1)-ого приближения. А также то, что, в отличие от ММК он сходится не при всех *λ*. Но, в отличие от ММК, нам не приходиться решать СЛАУ, поэтому алгоритм получается менее трудоёмким.