**Белорусский государственный университет**

**Факультет прикладной математики и информатики**

КСР - Вариант 3

по предмету “Функциональный анализ и интегральные уравнения”

Принцип сжимающих отображений

Выполнил :

Студент 2 курса 6 группы ФПМИ

Бусыгин Владимир

Преподаватель :

Чеб Елена Сергеевна

**Описание метода последовательных приближений для НУ**

Исходная задача : f(x) = 0.

Приведем f(x) к каноническому виду

Рассмотрим f(x) как оператор преобразования : , где

X - область изменения аргумента x, Y - область значений .

Теперь задача сводится к определению неподвижных точек и выбрав начальное приближение x0 в области, где существует единственный корень уравнения f(x) = 0, мы можем построить итерационный процесс :

Критерий останова итерационного процесса :

**Описание метода деления отрезка пополам**

Метод деления отрезка пополам применим в случае необходимости выделения промежутка из области значений функции (D(y)) на котором существовал бы единственный корень уравнения f(x) = 0.

На практике метод применялся следующим образом :

Первоначально мы определяем концы отрезка на котором мы будем искать необходимый нам отрезок, для своего уравнения я взял значения a = 100000 b = -100000 , x [a,b]

Дальше строим итерационный процесс следующим образом :

На каждой итерации выбираем точку x0 = (a + b)/2 , смещаем одну границу по правилу :

f(a)\*f(x0) < 0 , то b = x

 f(x0)\*f(b)<0, то a = x

Критерий остановки итерационного процесса : | f(x0) | < 0.5

**Вычисление априорной оценки количества итераций**

Априорную оценку количества итераций можно вычислить по формуле:

, где параметры q, m можно вычислить по следующим формулам : ,

m = 0.31338327998501825

q = 0.00045322

k = 12

**Код программы (Python)**

import matplotlib.pyplot as plt

import numpy as np

epsilon = pow(10, -4)

f = lambda x : 3\*x + np.log(1 + pow(x, 2)) + np.sin(x) + np.cos(x) - 7

phi = lambda x : -(np.log(1 + pow(x, 2)) + np.sin(x) + np.cos(x) - 7) / 3

x = np.linspace(-5, 5, 100)

plt.plot(x, phi(x))

plt.show()

def method(startValue):

countIteration = 1

x0 = startValue

x = phi(x0)

while(abs(phi(x) - x) >= epsilon):

countIteration += 1

x = phi(x)

return x, countIteration

def Dihotomia() :

x0, x1 = 1000000, -10000000

x = 0

while(True) :

x = (x0 + x1) / 2

if (abs(f(x0)) < 0.5):

break

elif (f(x0) \* f(x) < 0) :

x1 = x

else :

x0 = x

return x0, x1;

border1, border2 = Dihotomia()

print(**"Границы : ["**, border2, **","**,border1,**"]"**)

x0 = (border1 + border2)/2

print(**"x0 = "**, x0)

x, countIter = method(x0)

print(**"Решение : "**, x, **"**\n**Итерации : "**, countIter)

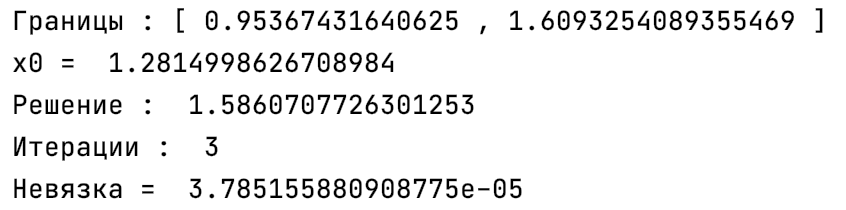
print(**"Невязка = "**, f(x))

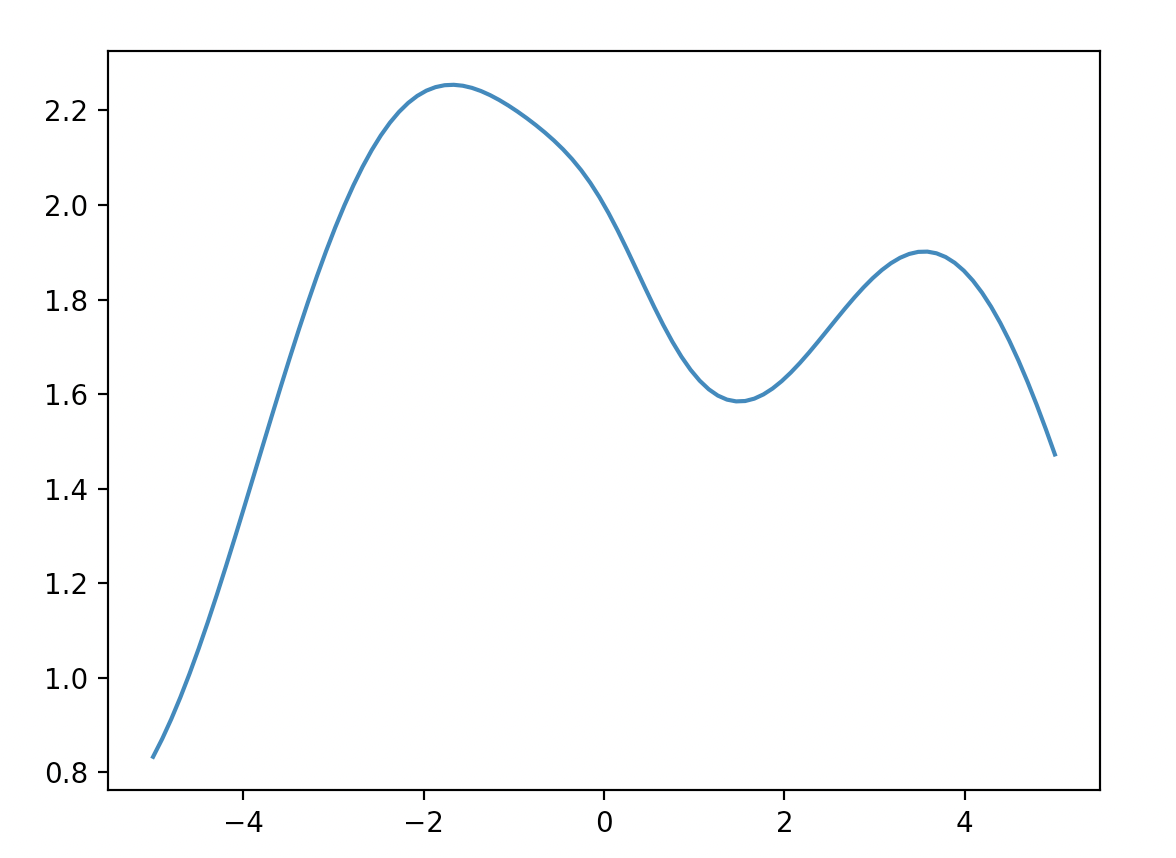
**Анализ полученных результатов**

**Входные данные :**



**Выходные данные :**

****

****

Посмотрев на невязку (f(x\*)) найденного решения, можно сказать, что полученное решение было достигнуто с необходимой точностью.

Сравнивая полученное количество итераций практическим путем с теоретическим вычислением количества итераций, можно убедиться, что итерационный метод сошелся за предполагаемое количество итераций.

**Вывод**

Мне удалось применить на практике принцип сжимающих отображений для нахождения одного из решений нелинейного уравнения.

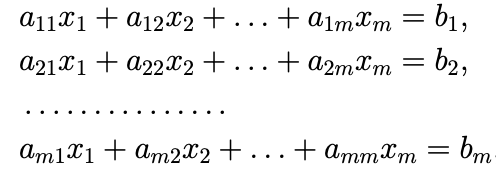
Метод крайне прост в реализации, однако требует предварительно нахождения начального значения. В своем случае я реализовывал поиск начального приближения, с помощью выделения промежутка, на котором бы располагался один из корней уравнения, методом деления отрезка пополам, начальное приближение выбиралось таким образом, что оно принадлежало бы выделенному промежутку.

Теоретически и практически мне удалось рассчитать количество итераций для нахождения приближенного значения корня нелинейного уравнения с заданной точностью.

Таким образом можно сказать, что данный способ применения принципа сжимающих отображений крайне удобен на практике и кроме того имеет быструю скорость сходимости.

**Описание метода последовательных приближений для СЛАУ**

Дана СЛАУ : Ax = b

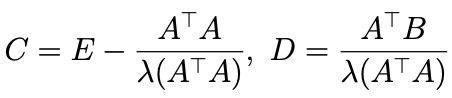


Если , то существует единственное решение системы, на практике же применим принцип сжимающих отображений для нахождения решения системы уравнений.

Для этого преобразуем систему так, чтобы отображение F(x) действовало из .

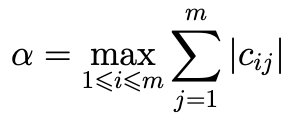
Ax = b

x = Cx + d , где матрицу C и вектор d удобно вычислить с помощью матричных формул :



Таким образом полученная матрица C будет иметь все собственные значения < 1, в связи с чем итерационный метод будет сходиться при любом начальном приближении x0.

Коэффициент сжатия можно вычислить по следующей формуле :



Априорная оценка количества итераций :

Вычисление коэффициента сжатия и априорной оценки было произведено в программе (*п. Анализ полученных данных*)

**Код программы (Python)**

import numpy as np

from numpy import linalg as lin

matrix\_A = np.array([ [9, 2, 1],[1, -7, 1],[1, 1, 9] ])

vector\_b = np.array([5, -6, -3])

matrix\_C = np.diag((1,1,1),0) - ( np.transpose(matrix\_A).dot(matrix\_A) / (np.amax(lin.eigvals(np.transpose(matrix\_A).dot(matrix\_A)))) )

vector\_d = np.transpose(matrix\_A).dot(vector\_b) / np.amax(lin.eigvals(np.transpose(matrix\_A).dot(matrix\_A)))

vector\_x0 = np.array([0, 0, 0])

epsilon = 10\*\*-3

def MPI\_FOR\_SLAY():

vector\_prev = vector\_x0

vector\_next = matrix\_C.dot(vector\_prev) + vector\_d

countIteration = 1

while(lin.norm(vector\_next - vector\_prev, ord = -np.inf) >= epsilon):

vector\_prev = vector\_next

vector\_next = matrix\_C.dot(vector\_prev) + vector\_d

countIteration += 1

return vector\_next, countIteration

def ApriorCountIterations():

return np.ceil(( (np.log(epsilon) + np.log( 1 - lin.norm(matrix\_C, ord = np.inf) ) + np.log(lin.norm(vector\_d, ord = np.inf)) )

/ np.log(lin.norm(matrix\_C, ord = np.inf)) ) - 1)

def CoefCompression():

return lin.norm(matrix\_C, ord = np.inf)

vector\_result, countIteartion = MPI\_FOR\_SLAY()

print(**"Количество итераций : "**, countIteartion)

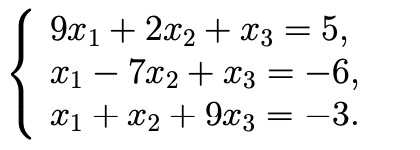
print(**"Решение : "**, vector\_result)

print(**"Коэффициент сжатия alfa : "**, CoefCompression())

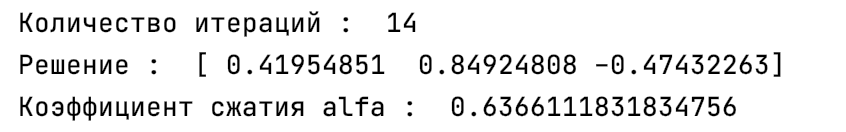
print(**"Априорная оценка количества итераций : "**, ApriorCountIterations())

**Анализ полученных результатов**

**Входные данные :**

****

**Выходные данные :**

****

****

**Вывод**

Действительно, полученное решение найденно с необходимой точностью

r = Ax - b = [ 0.00011015 0.00048934 -0.00010707].

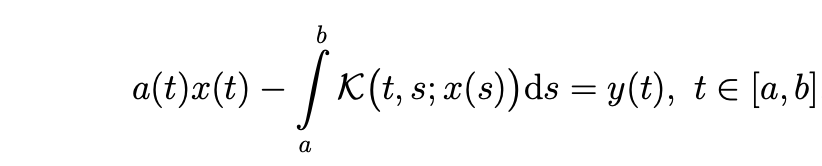
Поскольку матрица C , была сформирована способом описанным выше, то она будет иметь все собственные значения большие 0 и меньшие 1 :

[3.88578059e-16, 3.68373676e-01, 5.27796332e-01], а значит итерационный процесс будет сходится. Сравнивая полученное количество итераций на практике и априорную оценку, можно сказать,

что все полученные оценки сошлись, коэффициент сжатия < 1, что также удовлетворяет необходимому требованию.

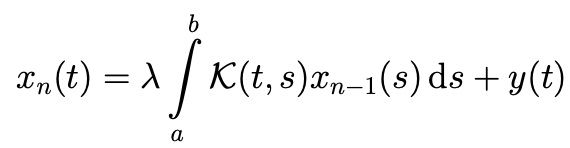
**Описание метода применение принципа сжимающих отображений для решения интегрального уравнения Фредгольма второго рода**

Имеется интегральное уравнение вида :



Если , то интегральное уравнение называют интегральным уравнением Фредгольма второго рода.

Построим итерационный метод для нахождения решения интегрального уравнения следующим образом :

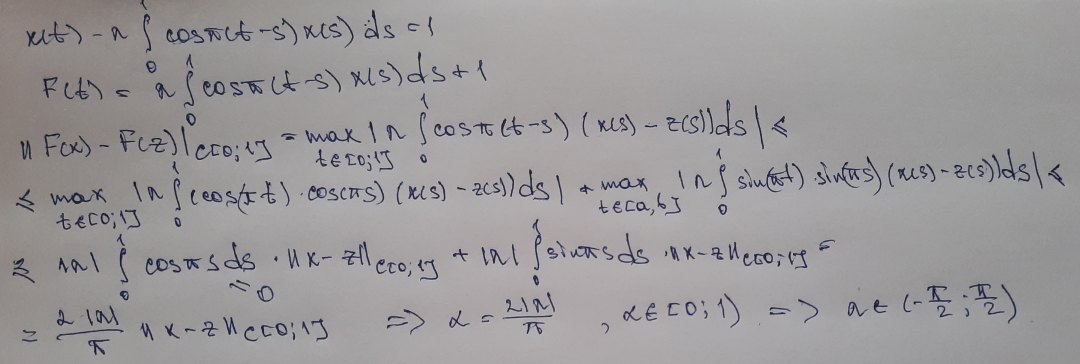


Критерий останова :

Решением интегрального уравнения будет являться x(t), так как линейное однородное уравнение всегда имеет решение , то в качестве начального приближения возьмем .

**Нахождение параметра**

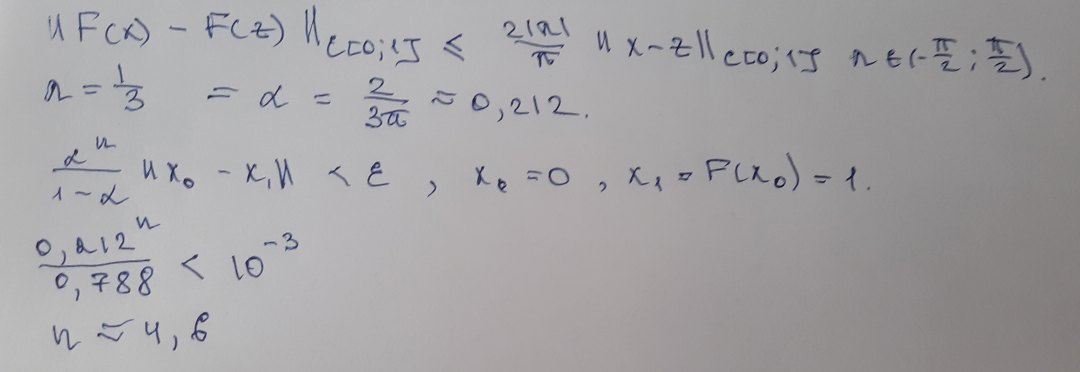
Чтобы выяснить при каком значении параметра можно применить метод последовательных приближений, необходимо проверить когда будет выполняться условие Липшица для нашего уравнения :



Таким образом нас удовлетворяет любое

В качестве начального значения параметра я взял .

**Вычисление априорной оценки количества итераций**



**Код программы (Python)**

from sympy import \*

lambda\_param = 1/3

a = 0

b = 1

epsilon = 10\*\*-3

t = symbols(**'t'**)

s = symbols(**'s'**)

def Fredgolm():

countIteration = 1

x = Lambda(t, 0) # start value x0() = 0

prev = Lambda(t, 0)

x = Lambda(t, lambda\_param \* integrate(cos(pi \* (t - s)) \* x(s), (s, a, b)) + 1)

while( (maximum(x(t) - prev(t), t, Interval(a, b)) > epsilon) or (maximum(prev(t) - x(t), t, Interval(a, b)) > epsilon) ):

prev = x

x = Lambda(t, lambda\_param \* integrate( cos(pi\*(t - s))\*x(s), (s, a, b)) + 1 )

countIteration += 1

return x, countIteration

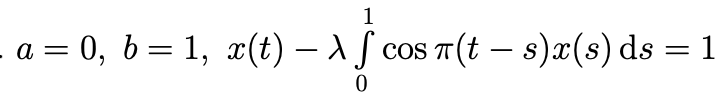
x, count = Fredgolm()

print(**"Решение = "**, x(t))

print(**"Количество итераций = "**, count)

**Анализ полученных результатов**

**Входные данные :**

****

**Выходные данные :**

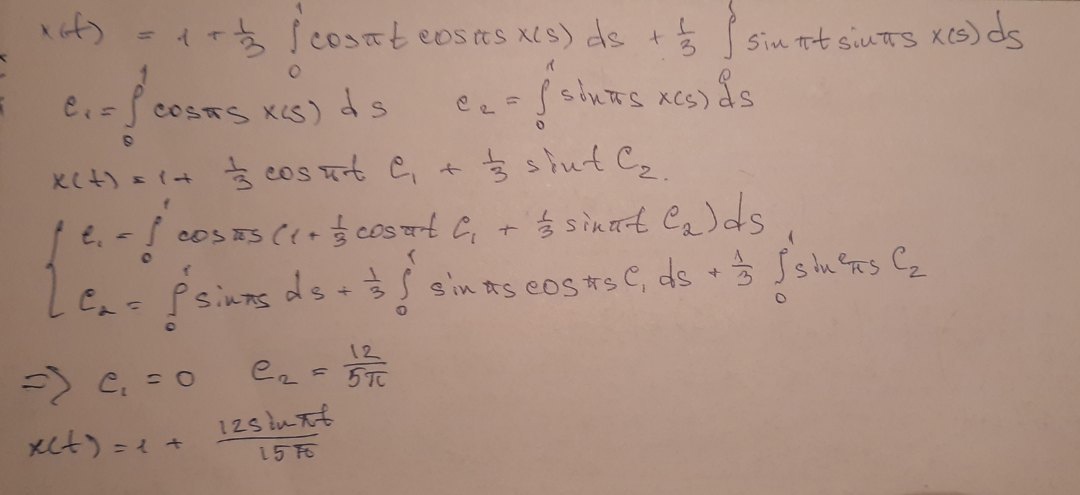
Решение = 0.33\*(0.78\*(pi\*sin(pi\*t)/2 - cos(pi\*t)/2)/pi + 1.0\*sin(pi\*t))/pi - 0.33\*(-1.0\*sin(pi\*t) - 0.39\*cos(pi\*t)/pi)/pi + 1

Количество итераций = 4

**Вывод**

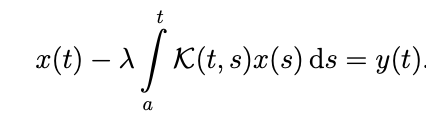
Действительно полученное решение при выбранном параметре является верным в пределах выбранной точности, кроме того можно сказать, что при выбранном параметре , который находится вне вычисленной области значений, итерационный процесс не сойдется, что говорит о том, что область значений была найдена верно.

Сравнивая полученное на практике количество итераций с вычисленной априорной оценкой, можно сказать, что они удовлетворяют друг другу, оценка на практике и теории была найдена верно.Также можно сравнить точное и полученное решение и убедится, что найденное решение соответсвует точному решению в пределах заданной точности.



**Описание метода применение принципа сжимающих отображений для решения интегрального уравнения Вольтера второго рода**

Имеется интегральное уравнение вида :



Построим последовательное приближение по правилу :

В качестве начального приближения возьмем

Критерий останова :

Поскольку в задании не указано, то возьмем его из 3-его задания :

**Код программы (Python)**

from sympy import \*

epsilon = 10\*\*-4

t = symbols(**'t'**)

s = symbols(**'s'**)

a = 0

b = 1

def Volter():

countIteration = 1

x = Lambda(t, 0)

prev = Lambda(t, 0)

x = Lambda(t, integrate( (t - s) \* x(s), (s, a, t)) + 1)

while ((maximum(x(t) - prev(t), t, Interval(a, b)) > epsilon) or (maximum(prev(t) - x(t), t, Interval(a, b)) > epsilon)):

prev = x

x = Lambda(t, integrate( (t - s) \* x(s), (s, a, t)) + 1)

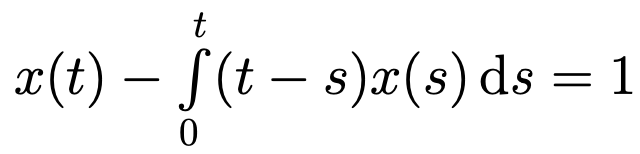
countIteration += 1

return x, countIteration

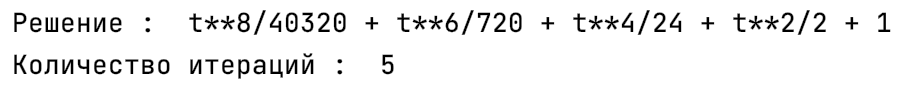
x, countIteration = Volter()print(**"Решение : "**, x(t))print(**"Количество итераций : "**, countIteration)

**Анализ полученных результатов**

**Входные данные :**

****

**Выходные данные :**

****

**Вывод**

Итерационный процесс практически идентичен случаю с интегральным уравнением Фредгольма, однако замеряя скорость выполнения первого и второго итерационного процесса, для достижения одной и той же точности приближенного решения требуется практически идентичное количество итераций, одного в случае с интегральным уравнением Вольтера 2-го рода процесс вычисления происходит заметно быстрее.

Как можно заметить приближенное решение уравнения было найдено с необходимой точностью.