**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа № 2**

по курсу «Численные методы».

Тема: «Решение нелинейных уравнений и систем нелинейных уравнений».

Студент: Шевчук П.В.

Группа: 80-304Б

Вариант: 13

Оценка:

Москва, 2019

Постановка задачи.

1. Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде

программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

1. Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений

в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

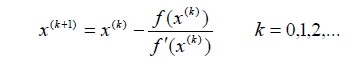
Описание методов.

Численное решение нелинейных (алгебраических или трансцендентных) уравнений заключается в нахождении значений x, удовлетворяющих (с заданной точностью) данному уравнению и состоит из следующих основных этапов:

1. Отделение (изоляция, локализация) корней уравнения.
2. Уточнение с помощью некоторого вычислительного алгоритма конкретного выделенного корня с заданной точностью.

Целью первого этапа является нахождение отрезков из области определения функции , внутри которых содержится только один корень решаемого уравнения. Иногда ограничиваются рассмотрением лишь какой-нибудь части области определения, вызывающей по тем или иным соображениям интерес. Для реализации данного этапа используются графические или аналитические способы.

Метод Ньютона (метод касательных). При нахождении корня уравнения (2.1) методом Ньютона, итерационный процесс определяется формулой



Для начала вычислений требуется задание начального приближения.

В качестве условия окончания итераций в практических вычислениях часто используется правило



Метод простой итерации. При использовании метода простой итерации уравнение заменяется эквивалентным уравнением с выделенным линейным членом

Решение ищется путем построения последовательности



Условия сходимости метода и оценка его погрешности определяются теоремой:

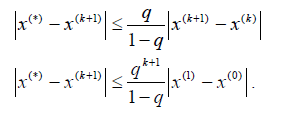
Пусть функция ϕ(x) определена и дифференцируема на отрезке [a,b]. Тогда если выполняются условия:



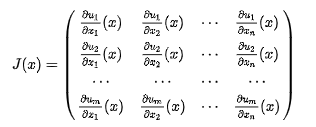
то уравнение (2.5) имеет и притом единственный на [a,b] корень точный корень;

к этому корню сходится определяемая методом простой итерации последовательность , начинающаяся с любого начальному х принадлежащему отрезку

При этом справедливы оценки погрешности.



Для систем нелинейных алгебраических уравнений всё аналогично. В методах надо использовать матрицу Якоби



Общая информация.

Данная работа состоит из нескольких модулей, которые позволяют решать нелинейные уравнения и их системы методами Ньютона и простых итераций. Полученные в ходе расчетов результаты сохраняются в отдельные файлы. Поскольку составленные программы могут обрабатывать любой корректный ввод (в том числе все варианты заданий из лабораторных работ), они могут служить удобным примером для реализации собственных решателей, применяясь для сравнения результатов. Что касается технических деталей реализации, все программы написаны на языке Python.

Запуск программы.

Чтобы воспользоваться программой, необходимо скомпилировать файл main.cpp и запустить полученный исполняемый файл, например, для g++ на Windows:

*python .\Lw2\_\*.py --eps 0.001 --output out1*

Выходные данные помещаются в файлы с префиксом out.

Результаты.

Вариант 1.

|  |  |
| --- | --- |
| 1 | EPS = 0.001  Iter: x = 0.42836568552905946  Steps = 3 |
|  | Newton: x = 0.42821161923307766  Steps = 2 |
| 2  ((x1\*\*2)/4) + (x2\*\*2) – 1  2\*x2-(math.e\*\*x1)-x1 | EPS = 0.001  Iter: x = [0.42492317 0.97717173]  Steps = 9 |
|  | Newton: x = [0.42442062 0.97684694]  Steps = 7 |

Выводы.

В ходе данной лабораторной работы были изучены алгоритмы Ньютона и метода простых итераций.

Метод Ньютона. Если точка x(0) выбрана на [a,b] так, что f(x(0) ) f ′′(x(0) ) > 0, то начатая с нее последовательность x(k)(k = 0,1,2,...) , определяемая методом Ньютона, монотонно сходится к корню уравнения.

Метод простой итерации. При использовании метода простой итерации уравнение заменяется эквивалентным уравнением с выделенным линейным членом x = ϕ(x)

Многомерный случай.

Метод Ньютона. Использование метода Ньютона предполагает дифференцируемость функцийf1(x), f2(x),…, fn(x) и невырожденность матрицы Якоби (det J(x(k))≠ 0). В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная.

Метод простой итерации. Процесс итерации сходится к единственному решению уравнения = ϕ(x)

Исходный код.

import argparse

import numpy as np

class Solver:

def \_\_init\_\_(self, eps, output\_name, log\_name):

self.eps = eps

self.out\_file = output\_name

self.log\_file = log\_name

if self.log\_file:

open(self.log\_file, 'w').close()

self.area = ( 0, 1)

self.x0 = 0.5

self.lmbd = self.calc\_lambda()

self.q = self.calc\_q()

self.k\_iter = 0

self.k\_newton = 0

self.check()

self.iter\_x = self.iter\_method()

self.newtons\_x = self.newtons\_method()

def check(self):

with open(self.log\_file, 'a') as f\_log:

f\_log.write(f'q = {self.q}\n')

f\_log.write(f'lambda = {self.lmbd}\n')

x = np.linspace(self.area[0], self.area[1], 10000)

y = [self.phi\_derivative(i) for i in x]

if all([i < 1 for i in y]):

f\_log.write('phi\' < 1\n')

else:

f\_log.write('phi\' >= 1\n')

if self.q < 1:

f\_log.write('q < 1\n')

else:

f\_log.write('q >= 1\n')

@staticmethod

def f(x):

return np.log(x+1)-2\*x+0.5

def phi(self, x):

return x - self.lmbd \* self.f(x)

def phi\_derivative(self, x):

return 1 - self.lmbd \* self.f\_derivative(x)

@staticmethod

def f\_derivative(x):

return (1/(x+1))-2

@staticmethod

def f\_2derivative(x):

return -1/(x\*\*2)

def calc\_q(self):

x = np.linspace(self.area[0], self.area[1], 10000)

y = [abs(self.phi\_derivative(i)) for i in x]

q = np.max(y)

return q

def calc\_lambda(self):

flag = None

x = np.linspace(self.area[0], self.area[1], 10000)

y = [self.f\_derivative(i) for i in x]

if all([np.sign(i) == -1 for i in y]):

flag = -1

elif all([np.sign(i) == 1 for i in y]):

flag = 1

else:

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write('Error: Derivative change sign\n')

exit(-1)

y = [abs(self.f\_derivative(i)) for i in x]

return flag / np.max(y)

def iter\_method(self):

x\_old = self.x0

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'Iter:\nx{self.k\_iter} = {x\_old}\n')

while True:

self.k\_iter += 1

x\_new = self.phi(x\_old)

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'x{self.k\_iter} = {x\_new}\n')

if abs(x\_new - x\_old) \* self.q / (1 - self.q) < self.eps:

return x\_new

else:

x\_old = x\_new

def newtons\_method(self):

x\_old = self.x0

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'Newton\'s:\nx{self.k\_iter} = {x\_old}\n')

while True:

self.k\_newton += 1

x\_new = x\_old - self.f(x\_old) / self.f\_derivative(x\_old)

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'x{self.k\_newton} = {x\_new}\n')

if abs(x\_new - x\_old) < self.eps:

return x\_new

else:

x\_old = x\_new

def print\_solution(self):

with open(self.out\_file, 'w') as f\_out:

f\_out.write(f'EPS = {self.eps}\n')

f\_out.write(f'Iter: x = {self.iter\_x}\n')

f\_out.write(f'Steps = {self.k\_iter}\n')

f\_out.write(f'Newton: x = {self.newtons\_x}\n')

f\_out.write(f'Steps = {self.k\_newton}\n')

def main():

parser = argparse.ArgumentParser()

parser.add\_argument('--eps', type=float, required=True, help='Accuracy')

parser.add\_argument('--output', required=True, help='File for answer')

parser.add\_argument('--log', help='Logging')

args = parser.parse\_args()

sol = Solver(args.eps, args.output, args.log)

sol.print\_solution()

if \_\_name\_\_ == "\_\_main\_\_": # python .\Lw2\_1.py --eps 0.001 --output out1

main()

2.

import argparse

import numpy as np

from numpy.linalg import norm, solve, det

from itertools import product

import math

class Solver:

def \_\_init\_\_(self, eps, output\_name, log\_name):

self.eps = eps

self.out\_file = output\_name

self.log\_file = log\_name

if self.log\_file:

open(self.log\_file, 'w').close()

#self.area = ((2.5, 3.25), (0.5, 0.75))

self.area = ((0, 1), (0.5, 1))

self.x0 = [0.5, 1]

self.k\_iter = 0

self.k\_newton = 0

self.lmbd = self.calc\_lambda()

self.q = self.calc\_q()

self.iter\_x = self.iter\_method()

self.newtons\_x = self.newtons\_method()

@staticmethod

def f1(x1, x2):

return ((x1\*\*2)/4) + (x2\*\*2) - 1

@staticmethod

def f2(x1, x2):

return 2\*x2-(math.e\*\*x1)-x1

@staticmethod

def f11(x1):

return x1/2

@staticmethod

def f12(x2):

return 2\*x2

@staticmethod

def f21(x1):

return -(math.e\*\*x1)-x1

@staticmethod

def f22(x2):

return 2

def phi1(self, x1, x2):

return x1 - (self.f1(x1, x2) \* self.lmbd[0, 0] + self.f2(x1, x2) \*

self.lmbd[0, 1])

def phi2(self, x1, x2):

return x2 - (self.f1(x1, x2) \* self.lmbd[1, 0] + self.f2(x1, x2) \*

self.lmbd[1, 1])

def phi11(self, x1, x2):

return 1 - (self.f11(x1) \* self.lmbd[0, 0] + self.f21(x1) \*

self.lmbd[0, 1])

def phi12(self, x1, x2):

return -(self.f12(x1) \* self.lmbd[0, 0] + self.f22(x2) \*

self.lmbd[0, 1])

def phi21(self, x1, x2):

return -(self.f11(x1) \* self.lmbd[1, 0] + self.f21(x1) \*

self.lmbd[1, 1])

def phi22(self, x1, x2):

return 1 - (self.f12(x1) \* self.lmbd[1, 0] + self.f22(x2) \*

self.lmbd[1, 1])

def phi\_derivative(self, x):

return np.array([[self.phi11(\*x), self.phi12(\*x)],

[self.phi21(\*x), self.phi22(\*x)]])

def j(self, x1, x2):

return [[self.f11(x1), self.f12(x1)], [self.f21(x1), self.f22(x2)]]

def calc\_lambda(self):

shape = len(self.area)

current\_j = self.j(\*self.x0)

inv\_j = np.array([solve(current\_j, i) for i in np.eye(shape)])

return np.transpose(inv\_j)

def calc\_q(self):

x1 = np.linspace(self.area[0][0], self.area[0][1], 100)

x2 = np.linspace(self.area[1][0], self.area[1][1], 100)

points = list(product(x1, x2))

vals = [norm(self.phi\_derivative(point), np.inf) for point in points]

q = np.max(vals)

return q

def iter\_method(self):

x\_old = self.x0

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'Iter:\nx{self.k\_iter} = {x\_old}\n')

while True:

self.k\_iter += 1

x\_new = np.array([self.phi1(\*x\_old), self.phi2(\*x\_old)])

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'x{self.k\_iter} = {x\_new}\n')

if norm(x\_new - x\_old, np.inf) \* self.q / (1 - self.q) <= self.eps:

return x\_new

else:

x\_old = x\_new

def newtons\_method(self):

shape = len(self.area)

x\_old = self.x0

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'Newton\'s:\nx{self.k\_iter} = {x\_old}\n')

while True:

current\_j = self.j(\*x\_old)

if det(current\_j) == 0:

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'Error: detJ({self.k\_newton} == 0)\n')

exit(-1)

self.k\_newton += 1

inv\_j = np.array([solve(current\_j, i) for i in np.eye(shape)])

x\_new = x\_old - np.transpose(inv\_j) @ np.array([self.f1(\*x\_old),

self.f2(\*x\_old)])

if self.log\_file:

with open(self.log\_file, 'a') as fl:

fl.write(f'x{self.k\_newton} = {x\_new}\n')

if norm(x\_new - x\_old, np.inf) <= self.eps:

return x\_new

else:

x\_old = x\_new

def print\_solution(self):

if self.log\_file:

with open(self.log\_file, 'a') as f\_log:

f\_log.write(f'q = {self.q}\n')

f\_log.write(f'Lambda:\n{self.lmbd}\n')

with open(self.out\_file, 'w') as f\_out:

f\_out.write(f'EPS = {self.eps}\n')

f\_out.write(f'Iter: x = {self.iter\_x}\n')

f\_out.write(f'Steps = {self.k\_iter}\n')

f\_out.write(f'Newton: x = {self.newtons\_x}\n')

f\_out.write(f'Steps = {self.k\_newton}\n')

def main():

parser = argparse.ArgumentParser()

parser.add\_argument('--eps', type=float, required=True, help='Accuracy')

parser.add\_argument('--output', required=True, help='File for answer')

parser.add\_argument('--log', help='Logging')

args = parser.parse\_args()

sol = Solver(args.eps, args.output, args.log)

sol.print\_solution()

if \_\_name\_\_ == "\_\_main\_\_":

main()