МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное

учреждение высшего образования

«Южный федеральный университет»

Институт компьютерных технологий и информационной безопасности

Кафедра интеллектуальных и многопроцессорных систем

Направление 01.04.02 Прикладная математика и информатика

**МАТЕМАТИЧЕСКИЕ МОДЕЛИ ПРОЦЕССОВ И СИСТЕМ**

Отчет по лабораторной работе №1

«Способы решения системы линейных алгебраических уравнений»

Выполнил:

магистрант группы КТмо1-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р.С. Булычев

подпись

Защита отчета состоялась

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Количество баллов\_\_\_\_\_\_\_\_\_\_\_\_

Проверил: д.т.н., профессор кафедры ИМС

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / А.В. Никитина /

подпись

Таганрог 2023

Вариант выполнения работы №3.

Задание 1.

Решить систему уравнений методом Гаусса и методом LU-разложения.

Решим систему методом Гаусса.

Краткое описание прямых методов расчёта СЛАУ

**1.** **Изучение последовательного алгоритма Гаусса решения систем линейных уравнений**

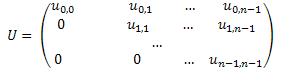
Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование эквивалентных). К числу таких преобразований относятся:

* Умножение любого из уравнений на ненулевую константу
* Перестановка уравнений
* Прибавление к уравнению любого другого уравнения системы

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – прямой ход метода Гаусса – исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду

http://www.hpcc.unn.ru/image.php?id=1195

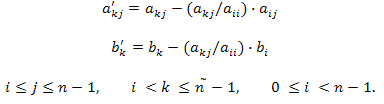
где матрица коэффициентов получаемой системы имеет вид



На обратном ходе метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной xn-1, после этого из предпоследнего уравнения становится возможным определение переменной xn-2 и т.д.

**1.1 Прямой ход метода Гаусса**

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации i метода производится исключение неизвестной i для всех уравнений с номерами k, большими i. Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (aki/aii), с тем, чтобы результирующий коэффициент при неизвестной xi в строках оказался нулевым. Вычисления, выполняемые над элементами матрицы A и вектора b, определяются следующими соотношениями.



На рисунке 1 представлена общая схема состояния данных на i-ой итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца i, уже являются нулевыми. На i-ой итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца i, расположенных ниже главной диагонали, путем вычитания строки i, умноженной на нужную ненулевую константу. После проведения (n-1) подобной итерации матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

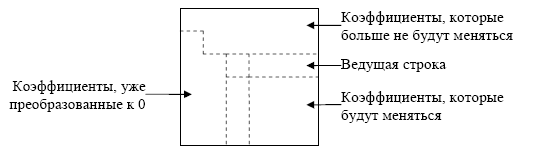


Рисунок 1- Общая схема состояния данных на i-ой итерации прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование ведущей, а диагональный элемент ведущей строки называется ведущим элементом. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент a(i,i) имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем – при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

http://www.hpcc.unn.ru/image.php?id=1199

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование метода главных элементов).

**1.2 Обратный ход алгоритма Гаусса**

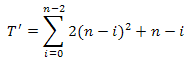
После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной xn-1, после этого из предпоследнего уравнения становится возможным определение переменной xn-2 и т.д. В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

http://www.hpcc.unn.ru/image.php?id=1200

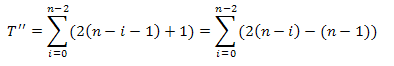
Определим вычислительную сложность метода Гаусса. В прямом ходе алгоритма для выбора ведущей строки на каждой итерации должно быть определено максимальное значение в столбце с исключаемой неизвестной. По мере исключения неизвестных количество строк и элементов в строках сокращается.

Текущее число элементов строки (включая правую часть), с которыми производятся действия сложения и вычитания (первый элемент без вычислений полагается равным нулю), равно (n-i), где i, – номер итерации прямого хода. Поскольку кроме выполнения двух операций (умножения и вычитания) с каждым элементом строки предварительно должен быть вычислен масштабирующий коэффициент aik/aii, общие затраты на выполнение действий в одной строке составят 2(n-i)+1 операций.

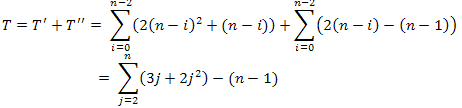
С учетом того, что на каждой итерации обрабатывается n-i строк, общее число операций в прямом ходе метода Гаусса определяется выражением



Для реализации обратного хода на каждой i-й итерации,(итерация для удобства присваиваем номера от n-2 до 0) необходимо произвести n-i-1 умножений, столько же вычитаний, а также одно деление для определения очередной неизвестной. Следовательно, общая вычислительная сложность обратного хода составит



Суммируя затраты на реализацию прямого и обратного хода, получаем



В последнем равенстве пределы и порядок суммирования изменены с учетом замены j=n-i.

Используя формулы суммирования,

http://www.hpcc.unn.ru/image.php?id=1204

получаем следующее соотношение для оценки вычислительной сложности метода Гаусса при его реализации в виде последовательного алгоритма:

http://www.hpcc.unn.ru/image.php?id=1205

Система из n линейных уравнений с n неизвестными имеет вид:

Метод Гаусса заключается в последовательном исключении неизвестных в каждом уравнении системы, начиная со второго, так называемый прямой ход, после чего находятся численные значения .

Для исключения переменной из всех уравнений системы, начиная со второго, необходимо ко второму уравнению системы прибавить первое, умноженное на коэффициент

а к третьему уравнению ‑ первое, умноженное на коэффициент

С остальными уравнениями поступаем аналогично.

Чтобы избежать деления на «0» примем, что коэффициенты при отличны от нуля. В случае, когда коэффициенты принимают нулевое значение, проводится либо сложение уравнений, либо их перестановка, что приводит к введению дополнительных условий и увеличивает количество операций.

В результате первой итерации мы получим СЛУ, в которой первое уравнение включает все неизвестные, а остальные – исключают :

Таким образом, для исключения из второго уравнения потребуется две операции («минус» и «деление») для нахождения компенсирующего множителя, а также операция умножения и операция сложения. Эта процедура повторяется раз. Следовательно, на первой итерации мы получим

количество арифметических операций.

Далее действуем аналогично, только с уравнениями, начиная со второго. Это будет вторая итерация для исключения неизвестного из остальных уравнений. Таким образом, после итерации получим:

Общее количество арифметических операций для приведения СЛУ общего вида (1) к виду (3) будет выражаться суммой числового ряда элементов, рассчитанных по формуле (2), от двух до n:

Существует два способа для нахождения неизвестных: метод подстановки и обратный ход по методу Гаусса. По второму способу СЛУ (3) приводится к виду:

По сути дела, данная процедура по количеству производимых арифметических действий эквивалентна прямому ходу, следовательно, общее количество операций равно удвоению полученного значения (4). Чтобы получить решение системы необходимо провести еще n делений:

Для реализации метода последовательной подстановки вычисляем из последнего уравнения:

Полученное значение подставляем в -е уравнение, находим значение , и так далее. С точки зрения количества операций, в этом случае мы имеем ряд арифметической прогрессии: 1, 3, 5, 7, …, сумма которой равна . Окончательно:

‑ для обратного хода:

‑ для последовательной подстановки:

Сравнительный анализ количества арифметических операций представлен в таблице 1.

Кол-во операций на обратном ходе

Кол-во операций для последовательной подстановки

Таким образом, для последующего анализа эффективности алгоритмов выбираем вариант последовательной подстановки.

Чтобы решить систему уравнений методом Гаусса, необходимо получить нули на главной и побочной диагонали.



B=

Проверим, является ли матрица невырожденной. Вычислим определитель матрицы 3×3.

=4\*(-1)\*5+2\*1\*(-1)+(-1)\*(-3)\*4-(1)\* (1)\* (1)-4\*1\*4-

2\*(-3)\*5=5

det A ≠0 => вывод матрица не вырождена.

Вычислим обратную матрицу. Допишем единичную матрицу.

C:\Users\User\AppData\Local\Microsoft\Windows\INetCache\Content.Word\определители_2.wmf  
  
Единичная матрица добавлена

100

010

001

Используя элементарные преобразования, 1-ю строку делим на 4.



Изменение единичной матрицы

0,25 0 0

0 1 0

0 0 1

К 2-й строки добавить 1-ю и умножить на3, к 3-й строке добавим 1-ю.



Деление единичной матрицы

0,25 0 0

0,75 1 0

0,25 0 1

2-ю строку делим на 0,5



0,25 0 0

1,5 2 0

0,25 0 1

От 1-й строки отнимаем 2-ю умноженную от 3-й строки отнимаем 2-й умноженной



Произведём деление предыдущей единичной матрицы

-0,5 -1 0

1,5 2 0

-6,5 -9 1

К1-й строке добавляем 3 строку умноженную на от строки 2отнименем 3-2 умноженную на

Более точное решение, применимо к СЛАУ

Получили единицу в 2-ом столбце разделив 2-ую строку на -11

Нашли единицу в 1-м столбце 3-ю и 4-ю строку поменяли местами

Постараемся привести первый столбец к нулю

Получили единицу в 2-ом столбце разделив 2-ую строку на -11

Обнулим второй столбец

Получаем единицу 3-м столбце делим 3-ю строку на

К 1-й строке добавляем 3\_ю строку. Умножим на 0,5 от 2-й строки отнимем 3-ю умножим на

Ответ:

Проверка: необходимо подставить все значения в начальное условие.

+2\*-=-+ = -1

3\*+-= = -1

-++5\*= -- = -8

Проверка произведена верно, следовательно уравнение методом Гаусса решено.

В результате проделанной работы решения методом Гаусса мы добились получения единичной матрицы.



В результате вычислена обратная матрица

Программная реализация задачи.

На маткаде (базовая часть)





Выбираем главный элемент для вычислений

**1.3 Реализация метода Гаусса в среде разработки mathcad**













Обратная Гаусса.







Нахождение обратной матрицы



**1.4 Программная реализация на Python**

**Реализация безоконного программного модуля**

import numpy as np  
  
def gauss(A, B):  
 # Прямой ход метода Гаусса  
 n = len(B)  
 for i in range(n):  
 # Поиск максимального элемента в столбце i  
 maxEl = abs(A[i][i])  
 maxRow = i  
 for k in range(i + 1, n):  
 if abs(A[k][i]) > maxEl:  
 maxEl = abs(A[k][i])  
 maxRow = k  
 # Обмен строками  
 for k in range(i, n):  
 tmp = A[maxRow][k]  
 A[maxRow][k] = A[i][k]  
 A[i][k] = tmp  
 tmp = B[maxRow]  
 B[maxRow] = B[i]  
 B[i] = tmp  
 # Приведение к верхнетреугольному виду  
 for k in range(i + 1, n):  
 c = -A[k][i] / A[i][i]  
 for j in range(i, n):  
 if i == j:  
 A[k][j] = 0  
 else:  
 A[k][j] += c \* A[i][j]  
 B[k] += c \* B[i]  
  
 # Обратный ход метода Гаусса  
 x = np.zeros(n)  
 for i in range(n - 1, -1, -1):  
 x[i] = B[i]  
 for j in range(i + 1, n):  
 x[i] -= A[i][j] \* x[j]  
 x[i] /= A[i][i]  
  
 return x  
  
# Определяем матрицу системы уравнений  
A = np.array([[4, 2, -1], [-3, 1, 1], [-1, 4, 5]])  
# Определяем столбец свободных членов  
B = np.array([-1, -1, -8])  
  
x = gauss(A, B)  
  
# Вывод результата  
print("Result:")  
print(x)

Результат отработанной программы с некоторым процентом погрешностей

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

matrix **=** np**.**array([[3.8, 6.7, **-**1.2, 5.2],

[6.4, 1.3, **-**2.7, 3.8],

[2.4, **-**4.5, 3.5, **-**0.6]]

withZero **=** np**.**array([[1,0,0, 1],

[0,0,1, 2],

[0,1,0, 3]], dtype**=**float)

Наивная реализация метода Гаусса приведения к треугольной форме. Сломается на матрицах, содержащих нуль на диагонали. Например, на матрице с такими коэффициентами:

array([[1., 0., 0.],

[0., 0., 1.],

[0., 1., 0.]])

Функция принимает на вход матрицу (N+1)xN - в последней колонке свободные члены. Функция меняет матрицу, переданную в аргументе, поэтому если хочется сохранить матрицу, то вызывать нужно с np.copy: gaussFunc(matrix.copy())

In [3]:

**def** makeTriangleNaive(matrix):

*# функция меняет матрицу через побочные эффекты*

*# если вам нужно сохранить прежнюю матрицу, скопируйте её np.copy*

**for** nrow, row **in** enumerate(matrix):

*# nrow равен номеру строки*

*# row содержит саму строку матрицы*

divider **=** row[nrow] *# диагональный элемент*

*# делим на диагональный элемент.*

row **/=** divider

*# теперь надо вычесть приведённую строку из всех нижележащих строчек*

**for** lower\_row **in** matrix[nrow**+**1:]:

factor **=** lower\_row[nrow] *# элемент строки в колонке nrow*

lower\_row **-=** factor**\***row *# вычитаем, чтобы получить ноль в колонке nrow*

*# все строки матрицы изменились, в принципе, можно и не возвращать*

**return** matrix

makeTriangleNaive(matrix**.**copy())

array([[ 1. , 1.76315789, -0.31578947, 1.36842105],

[-0. , 1. , 0.06800211, 0.49657354],

[ 0. , 0. , 1. , 0.09309401]])

Для нахождения решения нужно привести матрицу коэффициентов к диагональному виду. Тогда в последнем столбце будет находиться решение.

In [5]:

**def** makeIdentity(matrix):

*# перебор строк в обратном порядке*

**for** nrow **in** range(len(matrix)**-**1,0,**-**1):

row **=** matrix[nrow]

**for** upper\_row **in** matrix[:nrow]:

factor **=** upper\_row[nrow]

*# вычитать строки не нужно, так как в row только два элемента отличны от 0:*

*# в последней колонке и на диагонали*

*# вычитание в последней колонке*

upper\_row[**-**1] **-=** factor**\***row[**-**1]

*# вместо вычитания 1\*factor просто обнулим коэффициент в соотвествующей колонке.*

upper\_row[nrow] **=** 0

**return** matrix

m1 **=** makeTriangleNaive(np**.**copy(matrix))

m2 **=** makeIdentity(m1)

m2

Out[6]:

array([[ 1. , 0. , 0. , 0.53344344],

[-0. , 1. , 0. , 0.49024295],

[ 0. , 0. , 1. , 0.09309401]])

После приведения к диагональному виду корни находятся в последнем столбце.

roots **=** m2[:,**-**1]

roots

Out[7]:

array([0.53344344, 0.49024295, 0.09309401])

**Проверка решения**

Для проверки извлечём матрицу коэффициентов, умножим её справа на столбец корней и вычтем столбец свободных членов исходной матрицы: Ax - b. Результат должен оказаться близким к нулю.

In [8]:

coefs **=** matrix[:,:**-**1]

coefs

Out[8]:

array([[ 3.8, 6.7, -1.2],

[ 6.4, 1.3, -2.7],

[ 2.4, -4.5, 3.5]])

In [9]:

*# свободные члены в последнем столбце*

b **=** matrix[:,**-**1]

In [10]:

np**.**matmul(coefs, roots**.**T) **-** b

Out[10]:

array([ 0.00000000e+00, -4.44089210e-16, -2.22044605e-16])

**Решение СЛАУ одной функцией**

In [11]:

**def** gaussSolveNaive(A, b**=None**):

"""Решает систему линейных алгебраических уравнений Ax=b

Если b is None, то свободные коэффициенты в последней колонке"""

shape **=** A**.**shape

**assert** len(shape) **==** 2, ("Матрица не двумерная", shape) *# двумерная матрица*

A **=** A**.**copy()

**if** b **is** **not** **None**:

**assert** shape[0] **==** shape[1], ("Матрица не квадратная", shape)

**assert** b**.**shape **==** (shape[0],), ("Размерность свободных членов не соответствует матрица", shape, b**.**shape)

*# добавляем свободные члены дополнительным столбцом*

A **=** np**.**c\_[A, b]

**else**:

*# Проверяем, что квадратная плюс столбец*

**assert** shape[0]**+**1 **==** shape[1], ("Неверный формат матрицы", shape)

makeTriangleNaive(A)

makeIdentity(A)

**return** A[:,**-**1]

In [12]:

gaussSolveNaive(matrix)

Out[12]:

array([0.53344344, 0.49024295, 0.09309401])

In [13]:

gaussSolveNaive(matrix[:,:3], matrix[:,3])

Out[13]:

array([0.53344344, 0.49024295, 0.09309401])

Когда на диагонали встречается ноль, происходит деление на ноль. Оно не выбрасывается как исключение, вместо этого возвращается nan

In [14]:

gaussSolveNaive(withZero)

/usr/local/lib/python3.6/dist-packages/ipykernel\_launcher.py:9: RuntimeWarning: divide by zero encountered in true\_divide

if \_\_name\_\_ == '\_\_main\_\_':

/usr/local/lib/python3.6/dist-packages/ipykernel\_launcher.py:9: RuntimeWarning: invalid value encountered in true\_divide

if \_\_name\_\_ == '\_\_main\_\_':

Out[14]:

array([nan, nan, nan])

**Решение методом Гаусса с выбором главного элемента**

Для того, чтобы избежать проблем с делением на ноль, и вообще повысить устойчивость счета, используется метод Гаусса с выбором главного элемента.

В этом методе перед тем как делить на диагональный элемент среди всех строк, лежащих ниже, находится строка с максимальным по модулю элементом в нужной колонке.

In [15]:

**def** makeTrianglePivot(matrix):

**for** nrow **in** range(len(matrix)):

*# nrow равен номеру строки*

*# np.argmax возвращает номер строки с максимальным элементом в уменьшенной матрице*

*# которая начинается со строки nrow. Поэтому нужно прибавить nrow к результату*

pivot **=** nrow **+** np**.**argmax(abs(matrix[nrow:, nrow]))

**if** pivot **!=** nrow:

*# swap*

*# matrix[nrow], matrix[pivot] = matrix[pivot], matrix[nrow] - не работает.*

*# нужно переставлять строки именно так, как написано ниже*

*# matrix[[nrow, pivot]] = matrix[[pivot, nrow]]*

matrix[nrow], matrix[pivot] **=** matrix[pivot], np**.**copy(matrix[nrow])

row **=** matrix[nrow]

divider **=** row[nrow] *# диагональный элемент*

**if** abs(divider) **<** 1e-10:

*# почти нуль на диагонали. Продолжать не имеет смысла, результат счёта неустойчив*

**raise** ValueError("Матрица несовместна")

*# делим на диагональный элемент.*

row **/=** divider

*# теперь надо вычесть приведённую строку из всех нижележащих строчек*

**for** lower\_row **in** matrix[nrow**+**1:]:

factor **=** lower\_row[nrow] *# элемент строки в колонке nrow*

lower\_row **-=** factor**\***row *# вычитаем, чтобы получить ноль в колонке nrow*

**return** matrix

In [16]:

makeTrianglePivot(np**.**array([[1,0,0,1],

[0,0,1,2],

[0,1,0,3]

], dtype**=**float))

Out[16]:

array([[1., 0., 0., 1.],

[0., 1., 0., 3.],

[0., 0., 1., 2.]])

In [17]:

**def** gaussSolvePivot(A, b**=None**):

"""Решает систему линейных алгебраических уравнений Ax=b

Если b is None, то свободные коэффициенты в последней колонке"""

shape **=** A**.**shape

**assert** len(shape) **==** 2, ("Матрица не двумерная", shape) *# двумерная матрица*

A **=** A**.**copy()

**if** b **is** **not** **None**:

**assert** shape[0] **==** shape[1], ("Матрица не квадратная", shape)

**assert** b**.**shape **==** (shape[0],), ("Размерность свободных членов не соответствует матрица", shape, b**.**shape)

*# добавляем свободные члены дополнительным столбцом*

A **=** np**.**c\_[A, b]

**else**:

*# Проверяем, что квадратная плюс столбец*

**assert** shape[0]**+**1 **==** shape[1], ("Неверный формат матрицы", shape)

makeTrianglePivot(A)

makeIdentity(A)

**return** A[:,**-**1]

In [18]:

gaussSolvePivot(matrix)

Out[18]:

array([0.53344344, 0.49024295, 0.09309401])

В примере решается случайная система линейных уравнений с матрицей 100x100

In [19]:

N **=** 100

randomSle **=** np**.**random**.**rand(N, N)

randomV **=** np**.**random**.**rand(N)

Для начала решим "наивным" способом. Вероятность того, что на диагонали будет нуль, пренебрежимо мала.

In [20]:

randomRoots **=** gaussSolveNaive(randomSle, randomV)

randomRoots

Out[20]:

array([ 2.11416715, -1.30746648, -0.65419556, -3.51254613, 2.13898311,

3.22928076, 0.45769601, 2.43369704, 3.2711606 , 0.14570868,

0.68509975, 1.70555571, -1.05707612, 0.94090601, -0.87739547,

-0.70399065, -0.15676476, 1.00909638, -2.39858522, 0.62249626,

-1.77693207, 0.08585223, 0.21890165, 0.74606491, -0.61614036,

2.74852471, -1.35145162, -0.32323147, -0.0898949 , -0.74780049,

-1.34755001, 1.58825864, 0.56227854, -0.51789052, -2.28951741,

0.4885966 , -0.33649543, -1.33082582, -2.26453721, -0.46520173,

1.09681358, 0.37987709, 2.93641096, -0.22906293, -2.43658322,

2.16352784, -2.02093504, -1.66095716, -0.44670522, 1.87099628,

1.5777987 , 1.69613135, -2.01005121, -0.6260992 , 2.33762135,

1.87510222, 0.00690166, -0.60377963, -1.47735452, 1.21772367,

0.65785427, 1.99543894, -1.03656166, 1.24759644, -0.13939762,

-0.70099348, -1.37818259, -0.82149614, 0.36777295, 1.31172063,

-0.13168513, -1.02013248, -1.19356442, 0.40889253, -1.31970227,

-1.89835436, -1.00395977, -2.310658 , 2.29693745, 0.50310232,

2.09462163, 1.36730728, -0.62667526, -0.21577655, -0.951555 ,

-1.46197861, -2.30520095, -0.67303576, -0.93021393, -1.73937061,

1.03685333, 2.60066832, -0.16026882, 0.57375374, -2.1608723 ,

0.44546942, -1.90285726, -0.09570288, 2.60352348, 0.97680637])

In [21]:

randomRoots2 **=** gaussSolvePivot(randomSle, randomV)

Проверим решение: вычислим максимум модуля в разности Ax-b

In [22]:

diff **=** np**.**matmul(randomSle, randomRoots) **-** randomV

np**.**max(np**.**abs(diff))

Out[22]:

9.844347559351263e-13

In [23]:

diff **=** np**.**matmul(randomSle, randomRoots2) **-** randomV

np**.**max(np**.**abs(diff))

Out[23]:

1.9872992140790302e-14

В обоих случаях Ax практически равно b - корни найдены успешно. Но решение, найденное методом с выбором главного элемента, построило чуть более точное решение

Сравним найденное решение с решателем, который поставляется с numpy:

In [24]:

np\_roots **=** np**.**linalg**.**solve(randomSle, randomV)

np**.**max(np**.**abs(np**.**matmul(randomSle, np\_roots) **-** randomV))

Out[24]:

1.532107773982716e-14

In [25]:

max(abs(randomRoots **-** np\_roots)), max(abs(randomRoots2 **-** np\_roots))

Out[25]:

(2.374989094278135e-12, 5.81756864903582e-14)

Решения очень близкие. Встроенный решатель построил ещё более точное решение.

Сравнение времени счёта

In [26]:

**%timeit** -n10 -r 5 gaussSolveNaive(randomSle, randomV)

18.7 ms ± 2.66 ms per loop (mean ± std. dev. of 5 runs, 10 loops each)

In [27]:

**%timeit** -n10 -r 5 gaussSolvePivot(randomSle, randomV)

18.6 ms ± 367 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)

In [28]:

**%timeit** -n50 -r 7 np.linalg.solve(randomSle, randomV)

483 µs ± 39.1 µs per loop (mean ± std. dev. of 7 runs, 50 loops each)

Методы решения, написанные на чистом пайтоне, считают практически с одинаковой скоростью, и примерно в 40 раз медленее встроенного решателя. Ничего удивительного, встроенный решатель написан на Си.

Ниже представлен трюк, как можно приблизить скорость работы пайтоновского кода к Си-шному, если самые трудоёмкие части кода откомпилировать в машинный код компилятором numba

**Ускорение счёта**

Для начала обобщим метод решения, выделив функции приведения к треугольному виду и к диагональному виду в параметры.

In [29]:

**def** generalGauss(A,b, triangleFn**=**makeTrianglePivot, identityFn**=**makeIdentity):

"""Решает систему линейных алгебраических уравнений Ax=b

Если b is None, то свободные коэффициенты в последней колонке"""

shape **=** A**.**shape

**assert** len(shape) **==** 2, ("Матрица не двумерная", shape) *# двумерная матрица*

A **=** A**.**copy()

**if** b **is** **not** **None**:

**assert** shape[0] **==** shape[1], ("Матрица не квадратная", shape)

**assert** b**.**shape **==** (shape[0],), ("Размерность свободных членов не соответствует матрица", shape, b**.**shape)

*# добавляем свободные члены дополнительным столбцом*

A **=** np**.**c\_[A, b]

**else**:

*# Проверяем, что квадратная плюс столбец*

**assert** shape[0]**+**1 **==** shape[1], ("Неверный формат матрицы", shape)

A **=** triangleFn(A)

A **=** identityFn(A)

**return** np**.**array([ r[**-**1] **for** r **in** A ])

Проверим, что решение не изменилось

In [30]:

max(abs(generalGauss(randomSle, randomV, triangleFn**=**makeTriangleNaive, identityFn**=**makeIdentity) **-** randomRoots))

Out[30]:

0.0

In [31]:

**import** numba

Немного видоизменённый вариант функции makeTrianglePivot, адаптированный к возможностям компилятора numba.

Декоратор numba.njit предписывает транслировать функцию в чистый машинный код, который не обращается к интерпретатору пайтона. В общем случае это невозможно, но в данном случае у нас все вычисления идут только с numpy, а для этого пакета numba умеет вызывать Си-инетерфейсы для соответствующих операций - индексирования, присваивания, арифметики.

In [32]:

m **=** np**.**random**.**rand(3,4)

list(m)

Out[32]:

[array([0.35315579, 0.96583615, 0.53465876, 0.79239702]),

array([0.04410346, 0.04067085, 0.99929292, 0.50134466]),

array([0.10887379, 0.86640567, 0.04542141, 0.32615126])]

In [33]:

@numba**.**njit

**def** fastMakeTrianglePivot(matrix):

buf **=** np**.**zeros(matrix**.**shape[1])

**for** nrow **in** range(len(matrix)):

pivot **=** nrow **+** np**.**argmax(np**.**abs(matrix[nrow:, nrow]))

**if** pivot **!=** nrow:

matrix[nrow], matrix[pivot] **=** matrix[pivot], np**.**copy(matrix[nrow])

row **=** matrix[nrow]

divider **=** row[nrow] *# диагональный элемент*

**if** abs(divider) **<** 1e-10:

**raise** ValueError("Матрица несовместна")

row[nrow:] **\*=** 1**/**divider

row[nrow] **=** 1.0

**for** lr **in** range(nrow**+**1, len(matrix)):

lower\_row **=** matrix[lr]

factor **=** lower\_row[nrow]

np**.**multiply(factor, row, buf)

lower\_row **-=** buf

*# lower\_row -= factor\*row*

*# factor = matrix[lr, nrow]*

*# matrix[lr] -= factor\*row*

**return** matrix

In [34]:

@numba**.**njit

**def** fastMakeIdentity(matrix):

N **=** matrix**.**shape[0]

*# for nrow in range(len(matrix)-1,0,-1):*

*# root = matrix[nrow, -1]*

*# matrix[nrow:,-1] -= root\*matrix[nrow:,nrow]*

*# matrix[nrow:,nrow] = 0.0*

*# return matrix*

matrix **=** matrix**.**T

roots **=** matrix[**-**1]

**for** nrow **in** range(N**-**1,0,**-**1):

root **=** roots[nrow]

column **=** matrix[nrow]

roots[:nrow] **-=** root**\***column[:nrow]

column[:nrow] **=** 0.0

*# roots[nrow] = root*

**return** matrix**.**T

Сначала проверим, насколько выросла скорость от замены функции приведения к треугольному виду на скомпилированную Функцию вызываем два раза. В первом вызове jit-компилятор numba транслирует функцию fastMakeTrianglePivot в машинный код. Это долгая операция, поэтому результаты измерения времени будут недостоверными.

In [35]:

m1 **=** fastMakeTrianglePivot(np**.**random**.**rand(4,5))

m2 **=** fastMakeIdentity(m1**.**copy())

m1, m2

Out[35]:

(array([[ 1.00000000e+00, 9.77753041e-02, 7.47142889e-01,

4.63444137e-01, 9.15647911e-01],

[ 0.00000000e+00, 1.00000000e+00, 1.15719251e-02,

1.34530085e-01, -7.60750213e-02],

[ 0.00000000e+00, 0.00000000e+00, 1.00000000e+00,

8.61089532e-02, 4.43786691e-01],

[ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,

1.00000000e+00, -1.02261805e+02]]),

array([[ 1. , 0. , 0. , 0. ,

40.07040617],

[ 0. , 1. , 0. , 0. ,

13.57418043],

[ 0. , 0. , 1. , 0. ,

9.24944366],

[ 0. , 0. , 0. , 1. ,

-102.26180491]]))

In [36]:

fastRoots **=** generalGauss(randomSle, randomV, fastMakeTrianglePivot, fastMakeIdentity)

np**.**max(np**.**abs(fastRoots **-** randomRoots))

Out[36]:

2.3101520696400257e-12

In [37]:

**%timeit** -n10 -r3 generalGauss(randomSle, randomV, fastMakeTrianglePivot)

4.93 ms ± 80.6 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)

Благодаря компилятору время работы снизилось в 3 раза. Теперь заменим функцию приведения к диагональному виду на скомпилированную.

In [38]:

**%timeit** -n15 -r 5 generalGauss(randomSle, randomV, fastMakeTrianglePivot, fastMakeIdentity)

613 µs ± 49.8 µs per loop (mean ± std. dev. of 5 runs, 15 loops each)

Итого скорость выросла в 20 раз.

Проигрыш по сравнению с решателем на чистом Си/Фортране, меньше чем в 2 раза

Возьмём матрицу 1000 на 1000

In [39]:

N **=** 1000

randomSle **=** np**.**random**.**rand(N, N)

randomV **=** np**.**random**.**rand(N)

In [40]:

randomRoots **=** gaussSolveNaive(randomSle, randomV)

randomRoots2 **=** gaussSolvePivot(randomSle, randomV)

diffNaive **=** np**.**matmul(randomSle, randomRoots) **-** randomV

diffPivot **=** np**.**matmul(randomSle, randomRoots2) **-** randomV

np**.**max(np**.**abs(diffNaive)), np**.**max(np**.**abs(diffPivot))

Out[40]:

(2.966151768646341e-10, 7.12874204111813e-13)

Сравним найденные решения с решателем, который поставляется с numpy:

In [41]:

np\_roots **=** np**.**linalg**.**solve(randomSle, randomV)

np**.**max(np**.**abs(np**.**matmul(randomSle, np\_roots) **-** randomV))

Out[41]:

4.199973702156967e-13

In [42]:

max(abs(randomRoots **-** np\_roots)), max(abs(randomRoots2 **-** np\_roots))

Out[42]:

(3.828928285543043e-10, 1.4619416788264061e-12)

Решения близкие, но расстояние до них уже больше, чем для случая 100 на 100. Встроенный решатель построил ещё более точное решение.

Сравнение времени счёта

In [43]:

**%timeit** -n3 -r 1 gaussSolveNaive(randomSle, randomV)

2.55 s ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)

In [44]:

**%timeit** -n3 -r 1 gaussSolvePivot(randomSle, randomV)

2.56 s ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)

In [45]:

**%timeit** -n30 -r 7 np.linalg.solve(randomSle, randomV)

28.3 ms ± 2.69 ms per loop (mean ± std. dev. of 7 runs, 30 loops each)

Методы решения, написанные на чистом пайтоне, считают практически с одинаковой скоростью, но разрыв с Сишным/фортрановским решателем уже почти в тысячу раз. Причина в том, что метод Гаусса требует порядка (3) арифметических операций, и (2) операций аллокации временных векторов для вычитания. При увеличении размерности системы в 10 раз время счёта выросло в 100 раз. Можно предположить, что основной вклад в замедление счёта - это время на работу с памятью.

In [46]:

**%timeit** -n5 -r3 generalGauss(randomSle, randomV, fastMakeTrianglePivot)

712 ms ± 5.79 ms per loop (mean ± std. dev. of 3 runs, 5 loops each)

Благодаря компилятору время работы по-прежнему в 3 раза меньше, чем у функции на Python

In [47]:

**%timeit** -n10 -r 3 generalGauss(randomSle, randomV, fastMakeTrianglePivot, fastMakeIdentity)

278 ms ± 1.71 ms per loop (mean ± std. dev. of 3 runs, 10 loops each)

Выигрыш уже не столь значителен, меньше чем в 10 раз. И разница с встроенным решателем уже в 8 раз.

**2.1 Решение методом LU-разложения**

Решим систему методом LU-разложения.

Метод LU — разложения (декомпозиции) — один из способов решения системы линейных уравнений. Алгоритмы метода схожи с алгоритмами метода Гаусса.

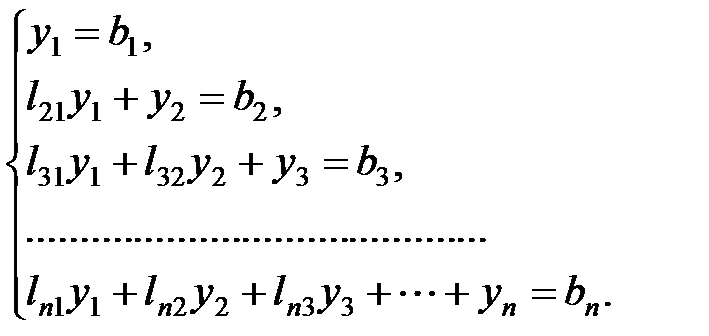
Суть метода состоит в том, чтобы представить исходную матрицу коэффициентов А как произведение двух треугольных матриц.

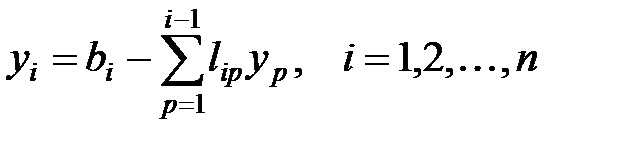
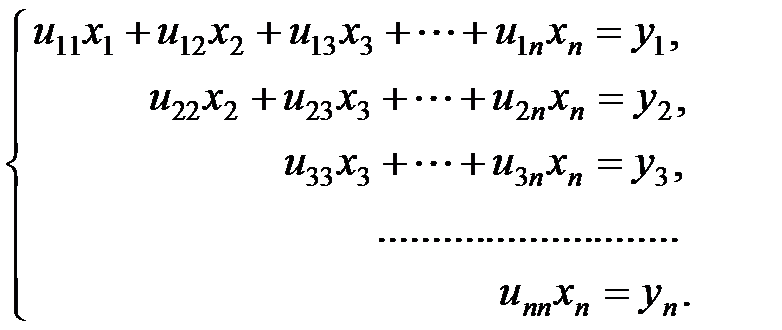
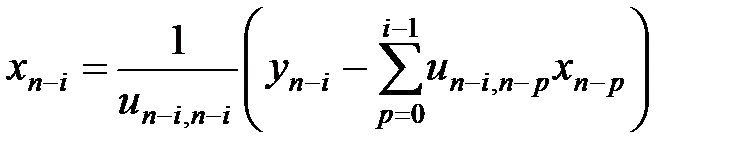
А = LU, где L — нижняя треугольная матрица с единичной диагональю, U — верхняя треугольная матрица. LU — разложение возможно, когда:  
— матрица А обратима;  
— главные миноры матрицы отличны от 0.

LU — разложение используют для решения систем линейных уравнений вида: Ах = b.

Т.к. А = LU, исходную систему можно представить в виде равенства: LUх = b. Если ввести вектор у = (у1, у2,...,уn)t, равенство можно представить как систему: Решение СЛАУ  
Т.е. решение системы Ах = b заключается в решении двух систем с треугольными матрицами: Lу = b, Uх = у.

На первом этапе решается система Lу = b. Т.к. L — нижняя треугольная матрица, система решается прямой подстановкой.

Запишем первую систему в виде:  
В первом уравнении вычисляем у1, во втором — у2, в третьем — у3 и т.д.

Общая формула:  
На втором этапе решается вторая система Uх = у способом обратной подстановки.  
Система имеет вид:  
Из последнего уравнения системы находим хn, из предпоследнего хn-1 и т.д., из первого находим х1.  
Общая формула для решения системы имеет вид:  
Быстро решать системы линейных уравнений методом LU — разложения можно с помощью онлайн калькулятора.

Сделаем проверку для понимания верности решения:

4\*(-0,26)+2\*(-0,58)+(-1)\*(-1,19)~ -1

-3\*(-0,26)+1\*(-0,58)+1\*(-1,19)~-1

(-1)\*(-0,26)+4\*(-0,58)+5\*(-1,19)~8

В каждом из ответов на решенную задачу можно наблюдать погрешность 0,01 для всех значений выражений СЛАУ. Это связано с погрешностями округления, тк математический модуль настроен на более точное решение поставленной задачи до 3 знаков после запятой.

**2.2 Решение задачи в среде разработки mathcad.**





















Получение обратной матрицы U





**2.3 Программный модуль LU разложения код на Python**

import numpy as np  
A = np.array([[4, 2, -1],  
 [-3, 1, 1],  
 [-1, 1, 1]])  
E1 = np.array([[1, 0, 0],  
 [-4, 1, 0],  
 [0, 0, 1]])  
E2 = np.array([[1, 0, 0],  
 [0, 1, 0],  
 [2, 0, 1]])  
  
E3 = np.array([[1, 0, 0],  
 [0, 1, 0],  
 [0, 1, 1]])  
  
E1\_inverse = np.linalg.inv(E1)  
E2\_inverse = np.linalg.inv(E2)  
E3\_inverse = np.linalg.inv(E3)  
  
U = E3.dot(E2).dot(E1).dot(A)  
L = E1\_inverse.dot(E2\_inverse).dot(E3\_inverse)  
  
print("\nStep 1 &amp; 2: Upper traingular matrix of A using elementary matrices:")#Верхняя треугольная матрица A с использованием элементарных матриц  
print(U)  
print("\nStep 1 &amp; 3: Lower traingular matrix of A using inverse elementary matrices:")#Нижняя треугольная матрица A с использованием обратных элементарных матриц  
print(L)  
  
U\_inverse = np.linalg.inv(U)  
L\_inverse = np.linalg.inv(L)  
  
b1 = np.array([[-1],  
 [-1],  
 [-8]]) # column vector  
  
c1 = L\_inverse.dot(b1)  
x1 = U\_inverse.dot(c1)  
print("\nStep 4a: Solve c1 given same left hand side matrix A but different right hand side b1:")#Решите c1, учитывая ту же левую часть матрицы A, но другую правую часть b1:  
print(c1)  
print("\nStep 5b: Solution x1 given same left hand side matrix A but different right hand side b1:")#Решение x1 дано с той же левой частью матрицы A, но с другой правой частью b1:  
print(x1)  
  
b2 = np.array([[28],  
 [22],  
 [-11]]) # column vector  
c2 = L\_inverse.dot(b2)  
x2 = U\_inverse.dot(c2)  
print("\nStep 4a: Solve c2 given same left hand side matrix A but different right hand side b2:")#Решите c2, учитывая ту же левую часть матрицы A, но другую правую часть b2  
print(c2)  
print("\nStep 5b: Solution x2 given same left hand side matrix A but different right hand side b2:")#Решение x2 дано с той же левой частью матрицы A, но с другой правой частью b2:  
print(x2)

**3.1 Решение методом прогонки**

Задание по условию.

3.2 Теоретические выкладки по практической работе

Метод прогонки представляет собой вариант метода Гаусса, примененный к специальным системам линейных алгебраических уравнений и учитывающий ленточную структуру матрицы системы. Пусть имеем СЛАУ со специальной трехдиагональной формой матрицы:

|  |  |
| --- | --- |
|  | (1.16) |

или в матричной форме: , где  – вектор неизвестных;  – вектор правых частей;  – квадратная  матрица:



Системы вида (1.16) возникают при конечно-разностной аппроксимации краевых задач математической физики, описываемых обыкновенными дифференциальными уравнениями второго порядка с постоянными и переменными коэффициентами, а также уравнениями в частных производных. Ставится задача разработать экономичные методы решения задач вида (1.16), число арифметических операций для которых пропорционально числу неизвестных. Таким методом для системы (1.16) является метод прогонки. Специфика матрицы  состоит в расположении ненулевых элементов, матрица  – разреженная матрица, из  элементов которой ненулевыми являются не более  элементов. Это позволяет получить для решения СЛАУ простые расчетные формулы.

Будем искать решение (1.16) в виде

|  |  |
| --- | --- |
|  | (1.17) |

с неопределенными коэффициентами . Выражение  подставим в (1.16):

,

c учетом (1.17) имеем

.

Это равенство имеет место для любых , если

, .

Отсюда получаем рекуррентные формулы для определения :

|  |  |
| --- | --- |
| ; | (1.18) |
| . | (1.19) |

Коэффициенты ,  называются прогоночными.

Если коэффициенты и  известны, а также известно , то, двигаясь справа налево (от  к ) последовательно определяем все . Задача нахождения  по формулам (1.18), (1.19) решается слева направо (от  к ). Начальные значения прогоночных коэффициентов  можно определить следующим образом. Полагаем в формуле (1.17) , имеем , а из первого уравнения (3.16) , откуда

|  |  |
| --- | --- |
| . | (1.20) |

Значение  определяется следующим образом. Полагаем в формуле (1.17) , имеем , а из последнего уравнения (3.18) –

,

откуда

|  |  |
| --- | --- |
| . | (1.21) |

Расчетные формулы (1.17) – (1.21) можно получить также из (1.16), если применить метод исключения Гаусса. Прямой ход метода заключается в том, что на первом шаге из всех уравнений системы (1.16) при помощи первого уравнения исключается , затем из преобразованных уравнений для  при помощи уравнения, соответствующего , исключается  и т.д. В результате получим одно уравнение относительно . На этом прямой ход метода прогонки заканчивается. На обратном ходе для  находятся .

Порядок счета в методе прогонки следующий:

1. исходя из значений , вычисленных по формулам (1.20), все остальные коэффициенты  для  определяются последовательно по формулам (1.18) и (1.19);
2. исходя из значения , рассчитанного по формуле (1.21), все остальные неизвестные ,  определяются последовательно по формуле (1.17).

Изложенный метод поэтому называется правой прогонкой.

Аналогично выводятся формулы левой прогонки:

|  |  |
| --- | --- |
|  | (1.22) |
|  | (1.23) |
| yi+1 = ξi+1yi + ηi+1, i = N-1, N-2, …, 0; y0 = η0. | (1.24) |

Здесь  находятся последовательно для значений ; ход вычислений – слева направо.

В случае, если необходимо найти только одно неизвестное, например,   или группу идущих подряд неизвестных, целесообразно комбинировать правую и левую прогонки. При этом получается метод встречных прогонок.

Произведем подсчет числа арифметических действий для метода правой прогонки. Анализ формул (1.17) – (1.21) показывает, что общее число арифметических операций есть . Коэффициенты  не зависят от правой части СЛАУ (1.16) и определяются только элементами  матрицы . Поэтому, если требуется решить серию задач (1.16) с различными правыми частями, то прогоночные коэффициенты  вычисляются только для первой серии. Для каждой последующей серии задач определяются только коэффициенты  и решение , причем используются ранее найденные .

На решение первой из серии задач расходуется  операций, а на решение каждой следующей задачи –  операций. Число арифметических операций, необходимое для решения СЛАУ (1.16) методом левой прогонки и методом встречных прогонок, такое же, т.е. . Метод правой прогонки будем называть корректным, если  при .

Решение  находится по рекуррентной формуле (1.17). Эта формула может порождать накопление ошибок округления результатов арифметических операций. Пусть прогоночные коэффициенты  и  найдены точно, а при вычислении  допущена ошибка , т.е. . При вычислениях с помощью формулы (1.17) мы получаем

|  |  |
| --- | --- |
| . | (1.25) |

Вычитая из (1.25) значение yi по формуле (1.17), имеем для погрешности  с заданным . Отсюда ясно, что если  по модулю больше единицы и если  достаточно велико, то вычисленное значение  будет значительно отличаться от искомого решения . В этом случае говорят, что алгоритм прогонки неустойчив.

*Определение*. Алгоритм прогонки называется устойчивым, если .

Условия корректности и устойчивости алгоритма правой прогонки определяются следующей теоремой.

**Теорема.** Пусть коэффициенты системы (1.16) действительны и удовлетворяют условиям:

,, , , , , ;

|  |  |
| --- | --- |
| , ; | (1.26) |
| , , | (1.27) |

причем хотя бы в одном из неравенств (1.26) и (1.27) выполняется строгое неравенство, т.е. матрица А имеет диагональное преобладание. Тогда для алгоритма (1.17) – (1.21) имеют место неравенства: , , т.е. алгоритм метода правой прогонки *корректен и устойчив*.

Условия (1.26) и (1.27) теоремы обеспечивают также корректность и устойчивость алгоритмов левой и встречных прогонок. Эти условия сохраняются и для случая системы (1.16) с комплексными коэффициентами .

Легко показать, что при выполнении условий (1.26) – (1.27) теоремы система (1.16) имеет единственное решение при любой правой части.

Выполним процедуру прямой прогонки.

A= B= x= обратная матрица

0 0 0

0 0 0 =

0 0 0 0

0 0

0 0 0

i=1

, =, =,

==1.000

= = = -1,

= = = 3

Реккурентная формула,

= + , = ,

=

= = = 1…

= = = 1

= +\* = 4 +(-1)\*1…0 = 3

= = - 0.667

= = - 0.667

Прямая прогонка вычисляется следующим образом:

,

= = = -1.00

= 2+(-1)\*(-0.667) = 2.667

Применяем обратную прогонку, используя последнюю строку матрицы (i=n)

При всех строк применяется

=

= = -1.0000

= \* += -0.667(-1.00)+(0.667) = 0.00

= \*-=1.000\*0.00+1.00 = 1.000

= \* + = -1.00\*1.000 +3 = 2.000

Ответ: Проверка:

= 2

= 1

= 0 = -1

**3.2 Разработка метода прогонки в среде разработки mathcad**

**Метод прогонки**



















































**3.3 Программная реализация на языке Python метода прогонки**

*'''  
Метод прогонки для СЛАУ из интервалов  
'''*import math  
  
class Element:  
 def \_\_init\_\_(self, numL, numR):  
 self.l = numL  
 self.r = numR  
  
 def \_\_add\_\_(self, other):  
 return Element(self.l + other.l, self.r + other.r)  
  
 def \_\_sub\_\_(self, other):  
 return Element(self.l-other.r, self.r-other.l)  
  
 def \_\_mul\_\_(self, other):  
 t1 = self.l \* other.l  
 t2 = self.l \* other.r  
 m1 = self.r \* other.l  
 m2 = self.r \* other.r  
 return Element(min(t1,t2), max(m1, m2))  
  
 def \_\_truediv\_\_(self, other):  
 t1 = self.l / other.l  
 t2 = self.l / other.r  
 m1 = self.r / other.l  
 m2 = self.r / other.r  
 return Element(min(t1,t2), max(m1, m2))  
  
def initial(a,b,c,d,M,R):  
 for i in range(M):  
 a.append(Element(0, 0))  
 b.append(Element(0, 0))  
 c.append(Element(0, 0))  
 d.append(Element(0, 0))  
   
 for i in range(1, M):   
 a[i] = Element((0.3\*math.sin(i+1)/V)-R, (0.3\*math.sin(i+1)/V)+R)  
  
 for i in range(M):  
 b[i] = Element(((10\*V)+((i+1)/V))-R, ((10\*V)+((i+1)/V))+R)  
 d[i] = Element((1.3+(i+1)/V)-R, (1.3+(i+1)/V)+R)  
  
 for i in range(M-1):   
 c[i] = Element((0.4\*math.cos(i+1)/V)-R, (0.4\*math.cos(i+1)/V)+R)  
  
 return a,b,c,d  
  
def progonka(a,b,c,d,M):  
 alpha = []  
 beta = []  
 x = []  
 for i in range(M):  
 x.append(Element(0, 0))  
  
 alpha.append(Element(-1,-1)\*c[0]/b[0])  
 beta.append(d[0]/b[0])  
  
 for i in range(1,M):   
 alpha.append((Element(-1,-1)\*c[i])/(b[i]+a[i]\*alpha[i-1]))  
 beta.append((d[i]-a[i]\*beta[i-1])/(b[i]+a[i]\*alpha[i-1]))  
  
 x[M-1] = beta[M-1]  
 for i in reversed(range(M-1)):  
 x[i] = alpha[i]\*x[i+1]+beta[i]  
  
 return alpha,beta,x  
  
def showEl(arrayEl):  
 for e in arrayEl:  
 print("{:.1e}".format((e.l+e.r)/2))  
  
def showX(arrayX):  
 for e in arrayX:  
 print("{:.5e} {:.5e} {:.5e}".format(e.l, (e.l+e.r)/2, e.r))  
  
if (\_\_name\_\_=='\_\_main\_\_'):   
   
 R = 0.01 #отклонение   
 M = int(input("Введите размерность: "))  
 V = 25.0  
 print("Вариант %s: " % (V))  
   
 '''  
 c - диагональ, лежащая над главной (нумеруется: [0;n-2])  
 b - главная диагональ матрицы A (нумеруется: [0;n-1])  
 a - диагональ, лежащая под главной (нумеруется: [1;n-1])  
 d - правая часть (столбец)  
 '''  
 a = []  
 c = []  
 b = []  
 d = []  
 a,b,c,d = initial(a,b,c,d,M,R)  
  
 alpha,beta,x = progonka(a,b,c,d,M)  
  
 print("\na")  
 showEl(a)  
  
 print("\nb")  
 showEl(b)  
  
 print("\nc")  
 showEl(c)  
  
 print("\nd")  
 showEl(d)  
  
 print("\nalpha")  
 showEl(alpha)  
  
 print("\nbeta")  
 showEl(beta)  
  
 print("\nx")  
 showX(x)

**Вывод:**

По результатам выполненной работы научились считать матрицы различными способами. Описаны программные модули для различных методов математических конструкции и применены реальные данные, полученные в результате решения всех предложенных уравнений. Исследованы закономерности изменений значений в зависимости от их точности.

Также данной работе был рассмотрен алгоритм Гаусса (прямой и обратный ход). Разработана программа, реализующая последовательный алгоритм на языке объектно-ориентированного программирования Python, рассчитано время выполнения последовательного алгоритма для разного размера матриц.

Также в данной работе рассмотренs различные библиотеки на языке Python для СЛАУ методом Гаусса.

Для оценки оптимальности разрабатываемых методов параллельных вычислений приведены широко используемые в теории и практике параллельного программирования основные показатели качества – ускорение (speedup), показывающее, во сколько раз быстрее осуществляется решение задач при использовании нескольких потоков, и эффективность (efficiency), которая характеризует долю времени реального использования потоков вычислительной системы.

При внимательном рассмотрении можно обратить внимание, что попытки повышения качества параллельных вычислений по одному из показателей (ускорению или эффективности) может привести к ухудшению ситуации по другому показателю, ибо показатели качества параллельных вычислений являются противоречивыми. Так, например, повышение ускорения обычно может быть обеспечено за счет увеличения числа процессоров, что приводит, как правило, к падению эффективности. И, обратно, повышение эффективности достигается во многих случаях при уменьшении числа процессоров. В предельном случае идеальная эффективность Ep(n)=1 легко обеспечивается при использовании одного процессора. Как результат, разработка методов параллельных вычислений часто предполагает выбор некоторого компромиссного варианта с учетом желаемых показателей ускорения и эффективности

**ПРИЛОЖЕНИЕ 1**