**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

Кафедра вычислительной математики

Жуковский Павел Сергеевич

Отчёт по лабораторной работе №3, вариант 5

(«Методы вычислений»)

Студента 2 курса 13 группы

Преподаватель

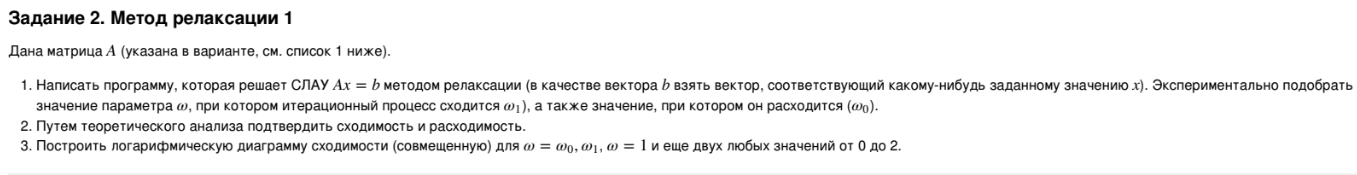
Бондарь Иван Васильевич

Минск 2020

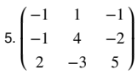
**Вариант:**

****

**Задание 2:**



Матрица:



Итак, я написал программу на языке Python, которая решает СЛАУ Ax = b методом релаксации, где в качестве вектора b я взял вектор (-7, 7, 28), который соответствует решению x = (7, 7, 7). Матрица A мне была дана по условию (картинка с ней прикреплена выше). Я также выбрал вот такое начальное приближение: x0 = (0, 0, 0).

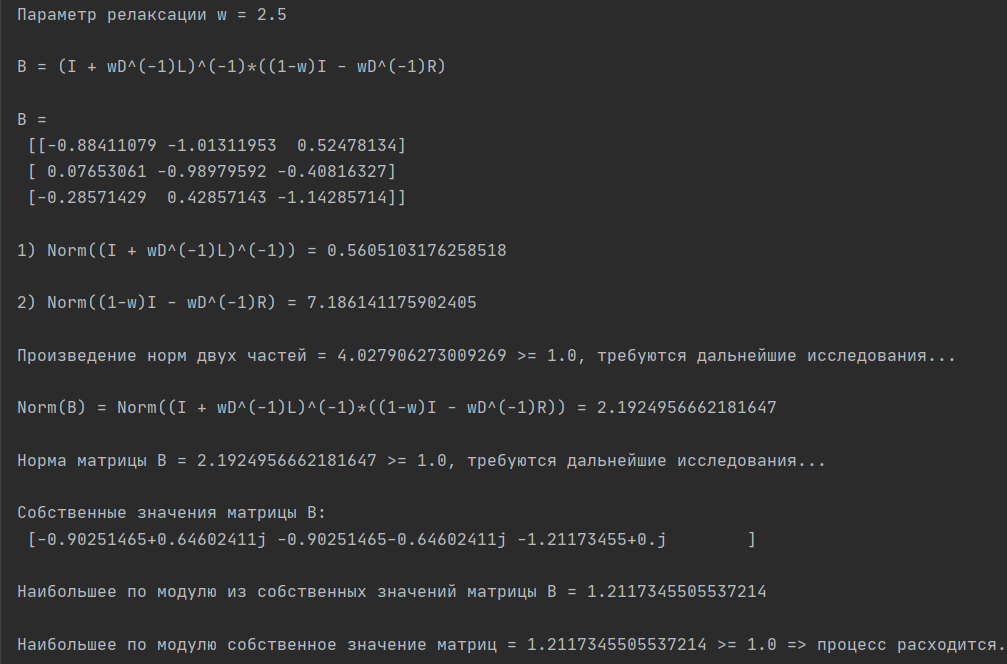
Мне требовалось экспериментально подобрать значение параметра w (он же параметр релаксации), при котором итерационный процесс сходится (w1), а также значение, при котором он расходится (w0).

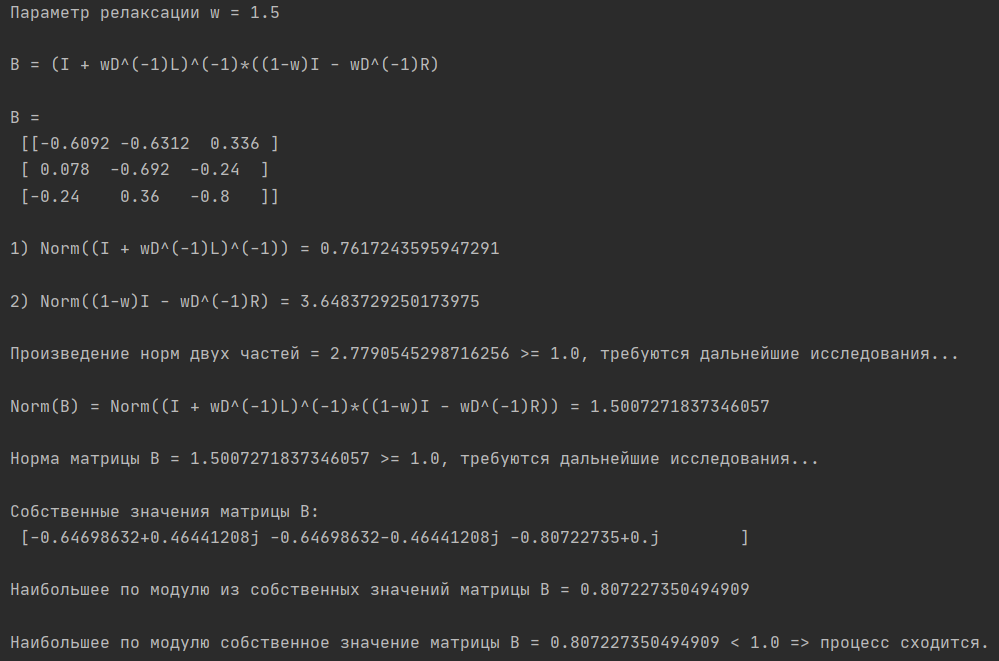
Для того, чтобы вообще определить, сходится или расходится процесс, я для начала проверял, не меньше ли единицы произведение норм двух частей матрицы B. Если не меньше единицы, ты я смотрел на норму самой матрицы B, которую получал путем произведения двух её частей (что это за части, будет видно в коде, где и реализован алгоритм релаксации). А если даже норма матрицы B не меньше единицы, то я искал собственные значения этой матрицы, выбирал из них наибольшее по модулю и смотрел, не меньше ли единицы оно. Если оно меньше единицы, то процесс сходится, а если – нет, то можно однозначно сказать, что процесс расходится.

Итак, вот исходный код моей программы:

import matplotlib.pyplot as plt  
import numpy as np  
import time  
  
# Лабораторная №3, Вариант №5, Задание №2, Матрица №5  
# Задание №2. Метод релаксации 1  
# Дана матрица A (указана в варианте, см. список 1 ниже).  
# 1. Написать программу, которая решает СЛАУ Ax = b методом релаксации (в качестве вектора b взять вектор,  
# соответствующий какому-нибуь заданному значению x). Экспериментально подобрать значение параметра w, при котором  
# итерационный процесс сходится w1), а также значение, при котором он расходится (w0).  
# 2. Путем теоретического анализа подтвердить сходимость и расходимость.  
# 3. Построить логарифмическую диаграмму сходимости (совмещенную) для w = w0, w1, w = 1 и еще двух любых значений от 0  
# до 2.  
  
# Номер итерационного процесса  
ProcessNum = 1  
  
# Требуемая точность (для итераций)  
Epsilon = 0.00000001  
  
# Размерность матрицы  
N = 3  
  
# Заданная матрица A  
A = np.array([[-1., 1., -1.],  
 [-1., 4., -2.],  
 [2., -3., 5.]])  
  
# Возьмём начальное приближение (0, 0, 0):  
X0 = np.array([0., 0., 0.])  
  
# В качестве вектора-ответа возьмём вектор (7, 7, 7), тогда вектор b = A\*(7, 7, 7) будет таким:  
b = np.array([-7., 7., 28.])  
  
# Значения w для экспериментов (w0 - не сходится, w1 - сходится, w2 - единица, w3 и w4 - любые от 0 до 2)  
w0 = 2.5  
w1 = 1.5  
w2 = 1.0  
w3 = 0.5  
w4 = 0.1  
  
# Посчитать норму невязки || A\*X = b || --> min  
def ResidualRate(X):  
 AX = np.dot(A, X) # A\*X  
 AX\_b = AX - b # A\*X - b  
 return np.linalg.norm(AX\_b) # || A\*X - b ||  
  
# Решение СЛАУ методом релаксации  
def RelaxationMethod(w, ResRateArr):  
 print("\n█████████████████████████████████████████████████████████████████████████████████████████\n")  
  
 global ProcessNum  
 print("ПРОЦЕСС №", ProcessNum, "\n")  
 ProcessNum += 1  
  
 StartTime = time.time()  
  
 print("Заданная точность Epsilon =", Epsilon, "\n")  
 print("Начальное приближение X0 =", X0, "\n")  
 print("Параметр релаксации w =", w, "\n")  
  
 L = np.tril(A, k=-1) # Нижнетреугольная матрица (на диагонали нужны нолики)  
  
 R = np.triu(A, k=1) # Верхнетреугольная матрица (на диагонали нужны нолики))  
  
 D = np.diag(np.diag(A)) # Диагональная матрица (со всеми нулями, т.к. это удобнее для последующего умножения)  
  
 ObrD = np.linalg.inv(D) # Матрица D^(-1), обратная матрице D  
  
 UnitMatrix = np.eye(N) # Единичная матрица размера NxN  
  
 # B(w) = (I + w\*D^(-1)\*L)^(-1)\*((1 - w)\*I - w\*D^(-1)\*R)  
 I\_wOBrDL\_Obr = np.linalg.inv(UnitMatrix + w\*np.dot(ObrD, L)) # 1) (I + wD^(-1)L)^(-1)  
 I\_1\_w\_wObrDR = (1 - w)\*UnitMatrix - w\*np.dot(ObrD, R) # 2) (1-w)I - wD^(-1)R  
  
 B = np.dot(I\_wOBrDL\_Obr, I\_1\_w\_wObrDR) # B(w) = (I + w\*D^(-1)\*L)^(-1)\*((1 - w)\*I - w\*D^(-1)\*R)  
 print("B = (I + wD^(-1)L)^(-1)\*((1-w)I - wD^(-1)R)\n\nB =\n", B, "\n")  
  
 NormI\_wObrDL\_Obr = np.linalg.norm(I\_wOBrDL\_Obr) # Норма первой части  
 print("1) Norm((I + wD^(-1)L)^(-1)) =", NormI\_wObrDL\_Obr, "\n")  
 NormI\_1\_w\_wObrDR = np.linalg.norm(I\_1\_w\_wObrDR) # Норма второй части  
 print("2) Norm((1-w)I - wD^(-1)R) =", NormI\_1\_w\_wObrDR, "\n")  
  
 BothPartsMult = NormI\_wObrDL\_Obr\*NormI\_1\_w\_wObrDR # Произведение обоих частей  
  
 if BothPartsMult < 1.: # Если произведение норм двух частей < 1, то  
 print("Произведение норм двух частей =", BothPartsMult, "< 1.0 => процесс сходится.\n")  
 else:  
 print("Произведение норм двух частей =", BothPartsMult, ">= 1.0, требуются дальнейшие исследования...\n")  
  
 NormB = np.linalg.norm(B) # Норма матрицы B  
 print("Norm(B) = Norm((I + wD^(-1)L)^(-1)\*((1-w)I - wD^(-1)R)) =", NormB, "\n")  
  
 if NormB < 1.: # Если норма самой матрицы B < 1, то  
 print("Норма матрицы B =", NormB, "< 1.0 => процесс сходится.\n")  
 else:  
 print("Норма матрицы B =", NormB, ">= 1.0, требуются дальнейшие исследования...\n")  
  
 EigenValuesB = np.linalg.eigvals(B) # Вектор, хранящий в себе собственные значения матрицы B  
 print("Собственные значения матрицы B:\n", EigenValuesB, "\n")  
 MaxEigenValueB = 0.  
 for i in range(EigenValuesB.size):  
 if abs(EigenValuesB[i]) > MaxEigenValueB:  
 MaxEigenValueB = abs(EigenValuesB[i])  
 print("Наибольшее по модулю из собственных значений матрицы B =", MaxEigenValueB, "\n")  
  
 if MaxEigenValueB < 1.: # Если максимальное по модулю собственное значение матрицы < 1, то  
 print("Наибольшее по модулю собственное значение матрицы B =", MaxEigenValueB, "< 1.0 => процесс сходится.\n")  
 else:  
 print("Наибольшее по модулю собственное значение матриц =", MaxEigenValueB, ">= 1.0 => процесс расходится.\n")  
  
 print("Процесс начал вычислительные итерации...\n")  
  
 # x\_k+1\_i = (1 - w)\*x\_k\_i + (w / a\_ii)\*(b\_i - Sum\_i-1\_j=1\_(a\_ij\*x\_k+1+j) - Sum\_n\_j=i+1\_(a\_ij\*x\_k\_j))  
 Xk = X0 # Вектор Xk - нужен для нахождения вектора Xk+1 в последующих итерациях  
 Xk\_1 = np.zeros(N) # Вектор Xk+1 - следующий вектор-ответ  
 IterationsAmount = 0 # Количество итераций  
 CurrResRate = ResidualRate(Xk) # Текущая невязка  
 while CurrResRate > Epsilon:  
 IterationsAmount += 1 # На каждой итерации приплюсовываем единицу к счетчику итераций  
 for i in range(N):  
 FirstSum = 0  
 for j in range(i):  
 FirstSum += (A[i, j] \* Xk\_1[j])  
  
 SecondSum = 0  
 for j in range(i + 1, N):  
 SecondSum += (A[i, j] \* Xk[j])  
  
 Xk\_1[i] = (1 - w) \* Xk[i] + (w / A[i, i]) \* (b[i] - FirstSum - SecondSum)  
  
 Xk = Xk\_1 # Говорим, что вектор Xk+1 в следующей итерации будет просто Xk  
 CurrResRate = ResidualRate(Xk) # Текущая невязка  
 ResRateArr.append(CurrResRate) # Добавляем текущую невязку в список невязок для графика  
  
 print("После", IterationsAmount, "итерации был подобран X =", Xk, "\n")  
  
 print("Общее время работы процесса: %s seconds" % (time.time() - StartTime), "\n")  
  
 return IterationsAmount # По завершении процесса возвращаем количество итераций, которое нам понадобилось  
  
  
print("\nМатрица A =\n", A, "\n")  
print("Вектор b =\n", b)  
# Значения t, в которых будем хранить времена работы всех процессов  
ResRateArr1 = [] # Список ординат (норм невязки на разных итерациях) для графика 1-ого процесса  
IterAmount1 = RelaxationMethod(w0, ResRateArr1)  
IterArr1 = np.arange(1, IterAmount1 + 1) # Массив абсцисс (количества итераций) для графика 1-ого процесса  
ResRateArr2 = [] # Список ординат (норм невязки на разных итерациях) для графика 2-ого процесса  
IterAmount2 = RelaxationMethod(w1, ResRateArr2)  
IterArr2 = np.arange(1, IterAmount2 + 1) # Массив абсцисс (количества итераций) для графика 2-ого процесса  
ResRateArr3 = [] # Список ординат (норм невязки на разных итерациях) для графика 3-его процесса  
IterAmount3 = RelaxationMethod(w2, ResRateArr3)  
IterArr3 = np.arange(1, IterAmount3 + 1) # Массив абсцисс (количества итераций) для графика 3-его процесса  
ResRateArr4 = [] # Список ординат (норм невязки на разных итерациях) для графика 3-его процесса  
IterAmount4 = RelaxationMethod(w3, ResRateArr4)  
IterArr4 = np.arange(1, IterAmount4 + 1) # Массив абсцисс (количества итераций) для графика 4-ого процесса  
ResRateArr5 = [] # Список ординат (норм невязки на разных итерациях) для графика 5-ого процесса  
IterAmount5 = RelaxationMethod(w4, ResRateArr5)  
IterArr5 = np.arange(1, IterAmount5 + 1) # Массив абсцисс (количества итераций) для графика 5-ого процесса  
plt.semilogy(IterArr1, ResRateArr1, label='w0')  
plt.semilogy(IterArr2, ResRateArr2, label='w1')  
plt.semilogy(IterArr3, ResRateArr3, label='w2')  
plt.semilogy(IterArr4, ResRateArr4, label='w3')  
plt.semilogy(IterArr5, ResRateArr5, label='w4')  
plt.xlabel("Номер итерации")  
plt.ylabel("Норма невязки на этой итерации")  
plt.legend()  
plt.show()

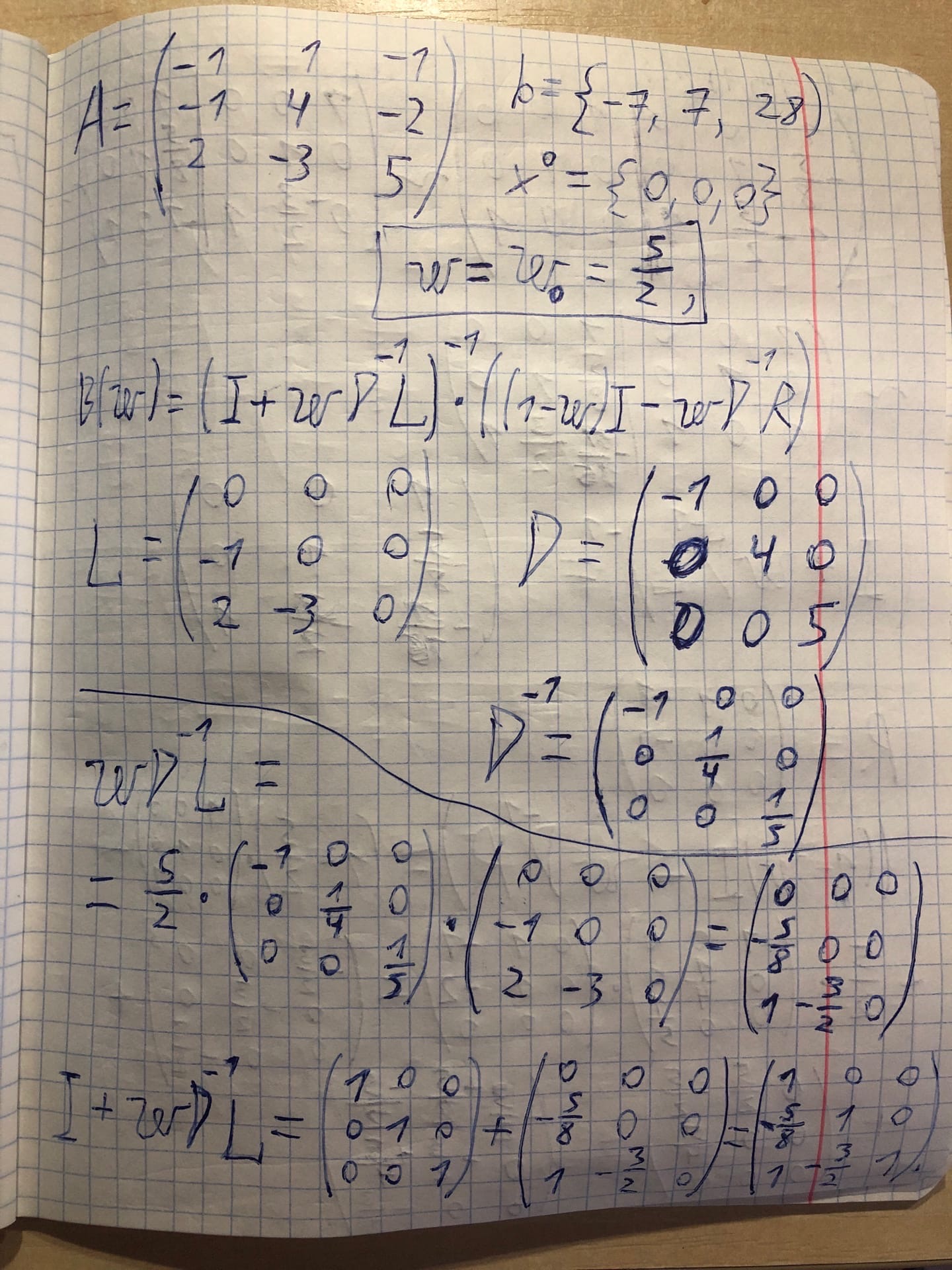
Делая различные эксперименты, я выяснил, что при w0 = 2.5 процесс расходится, а при w1 = 1.5 процесс сходится:

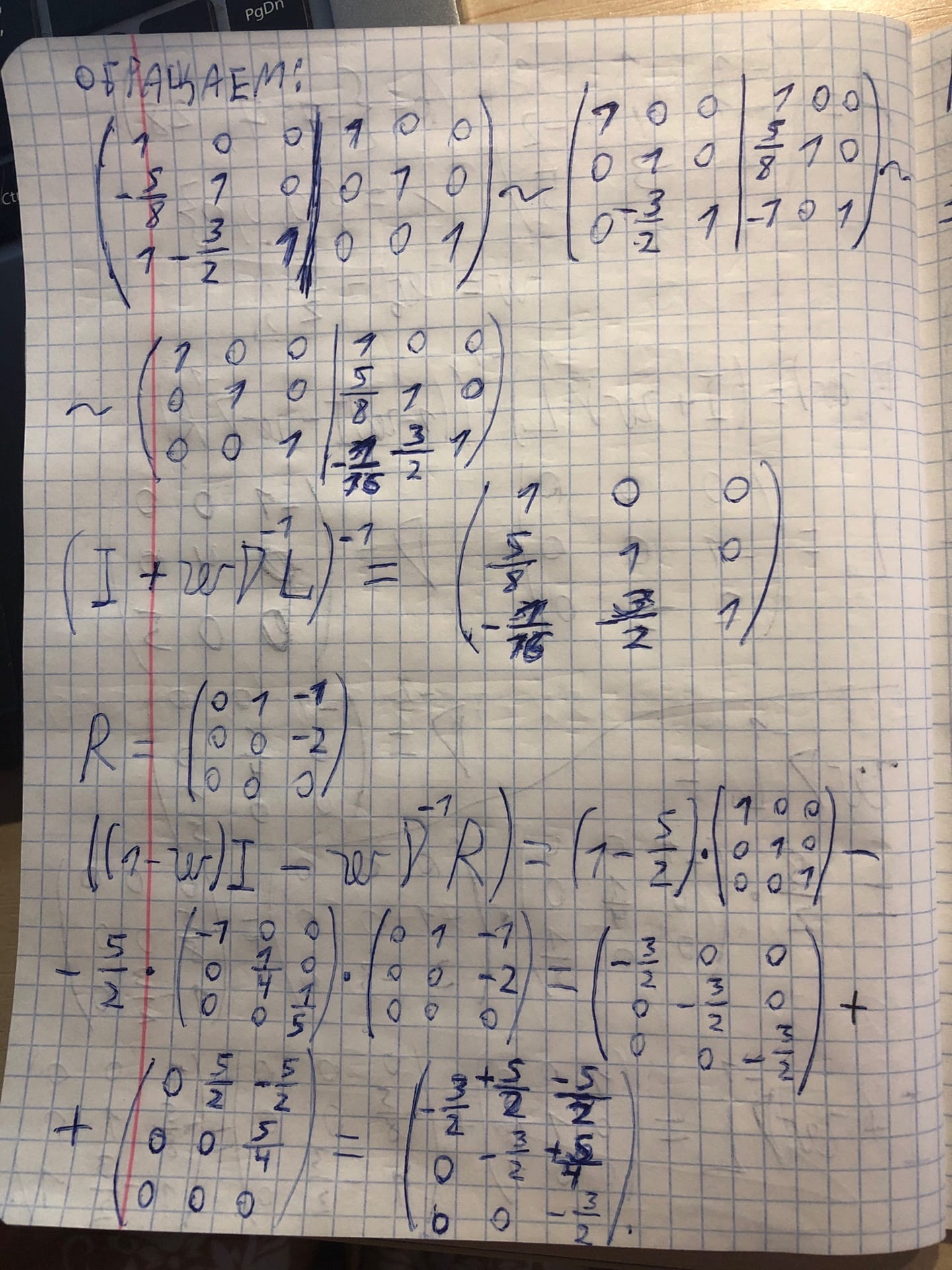


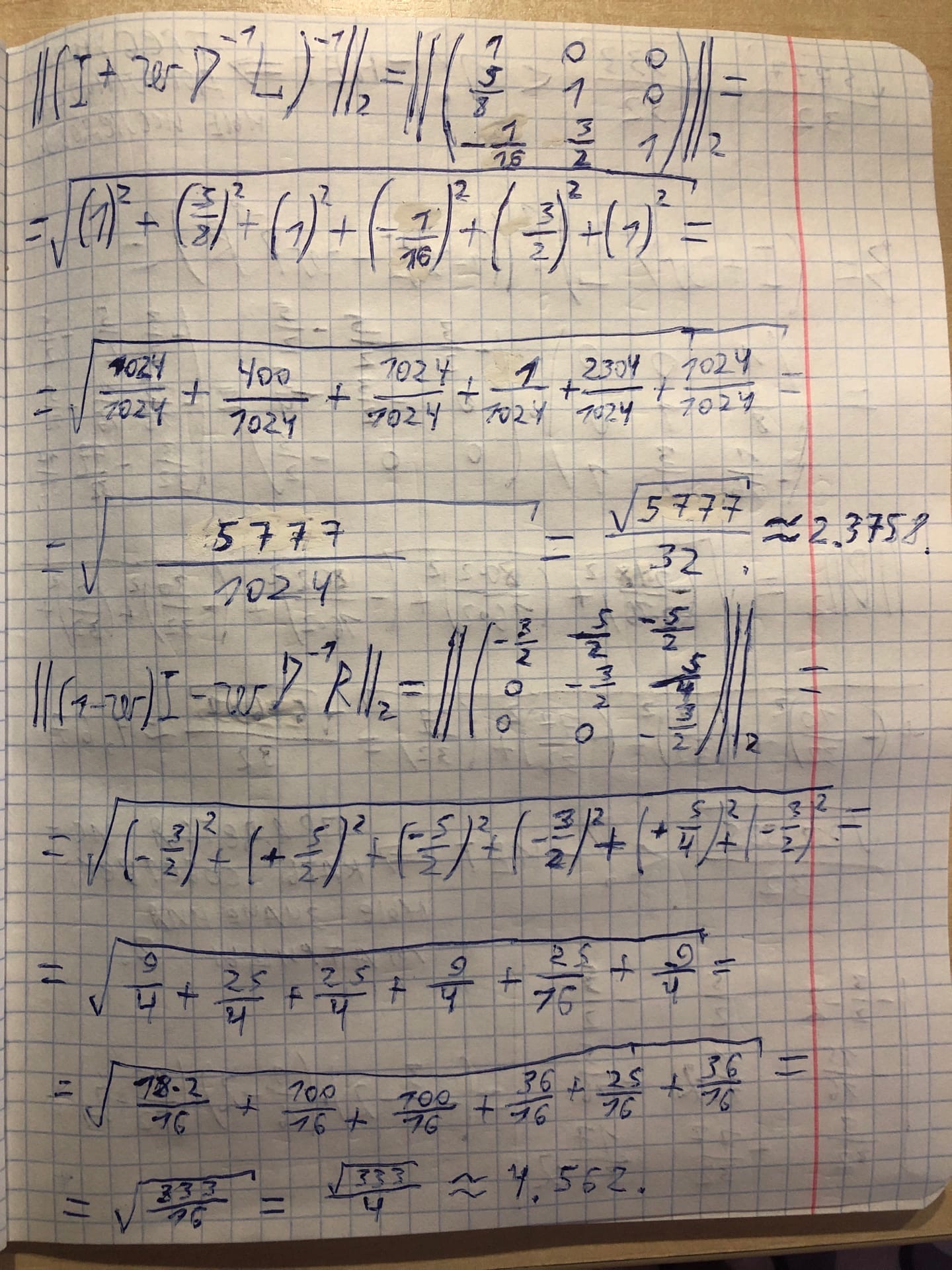


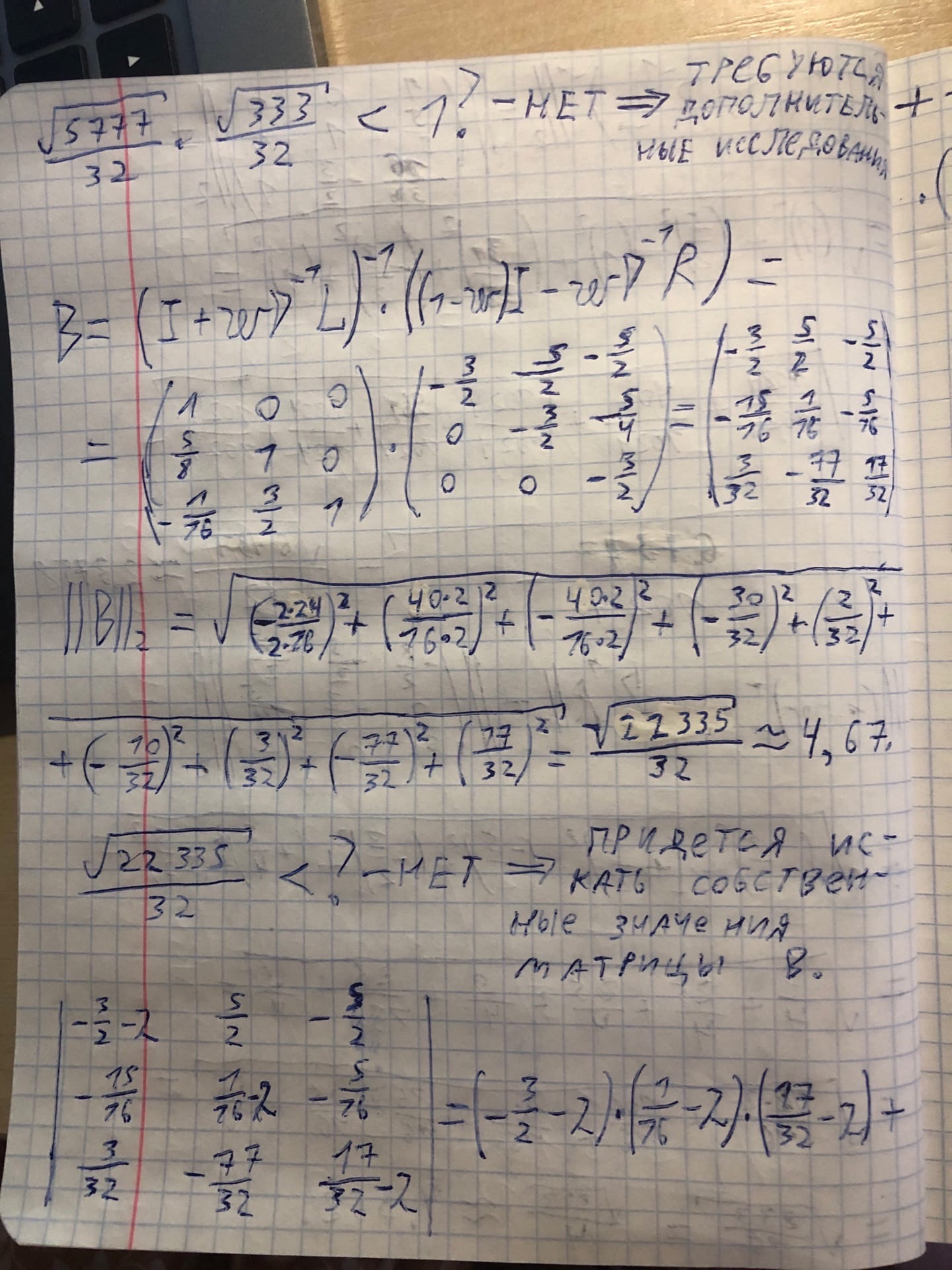
Позже я доказал расходимость w0 и сходимость w1 теоретически.

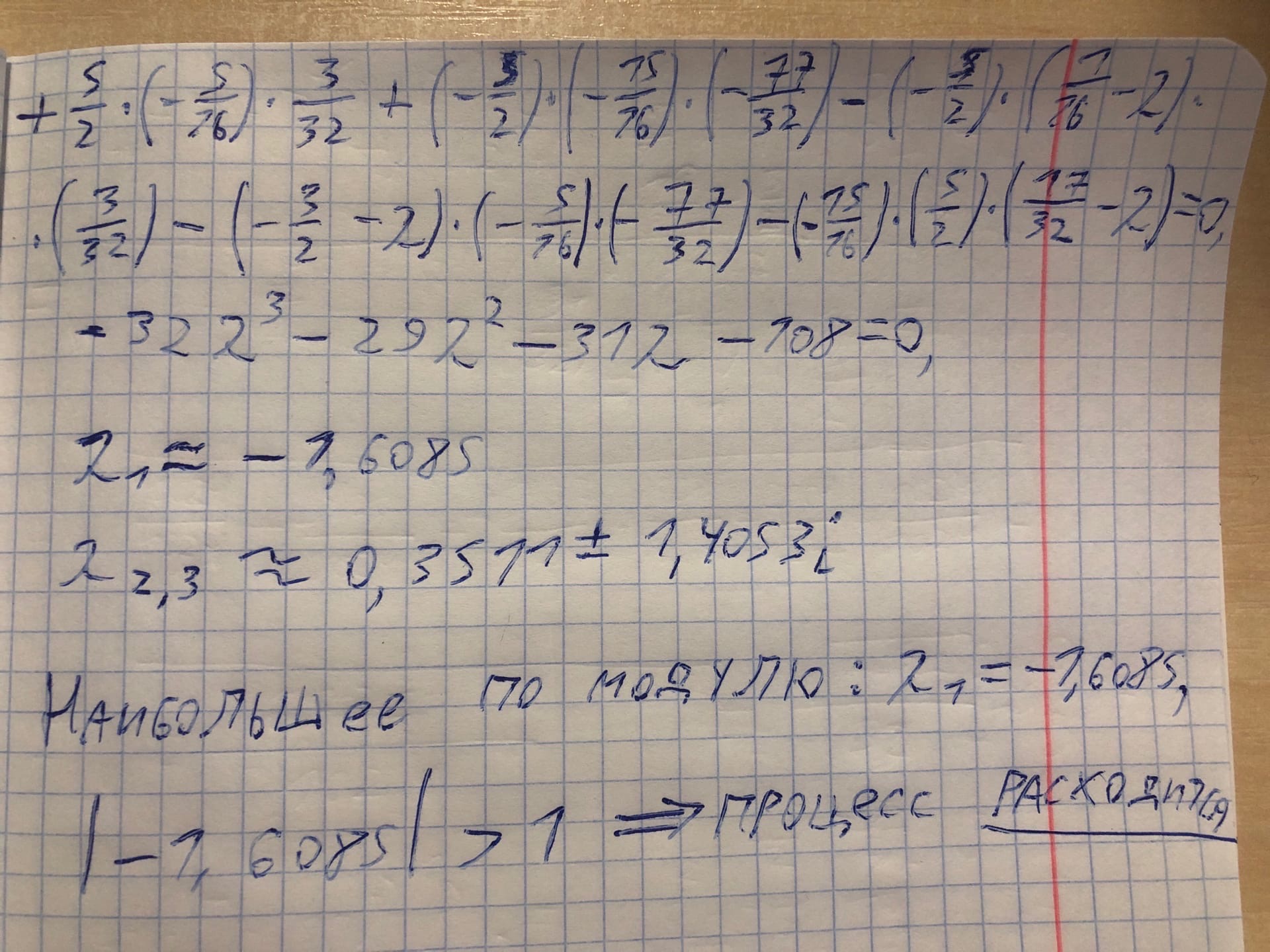
Доказательство расходимости процесса при w0 = 2.5:





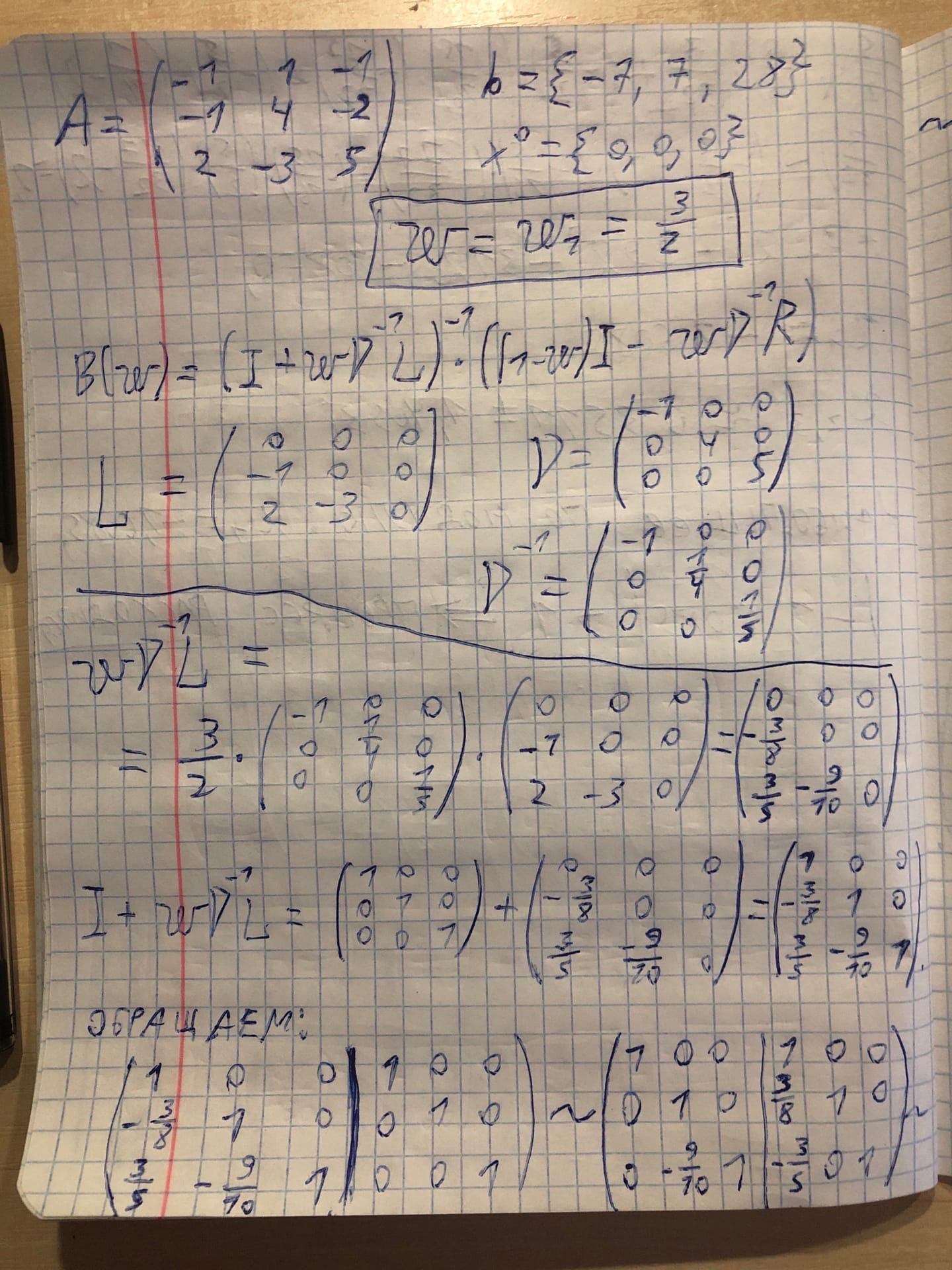


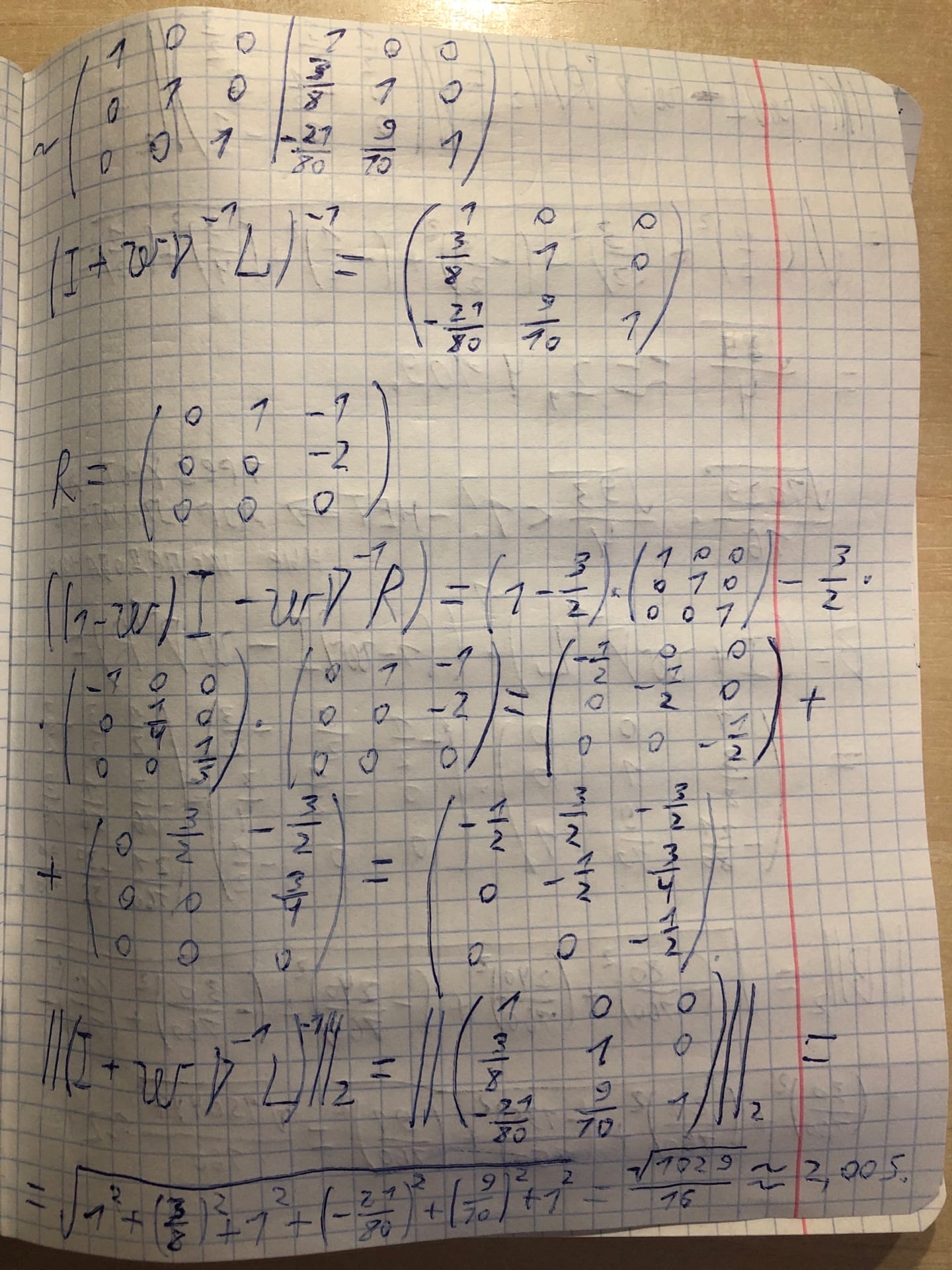


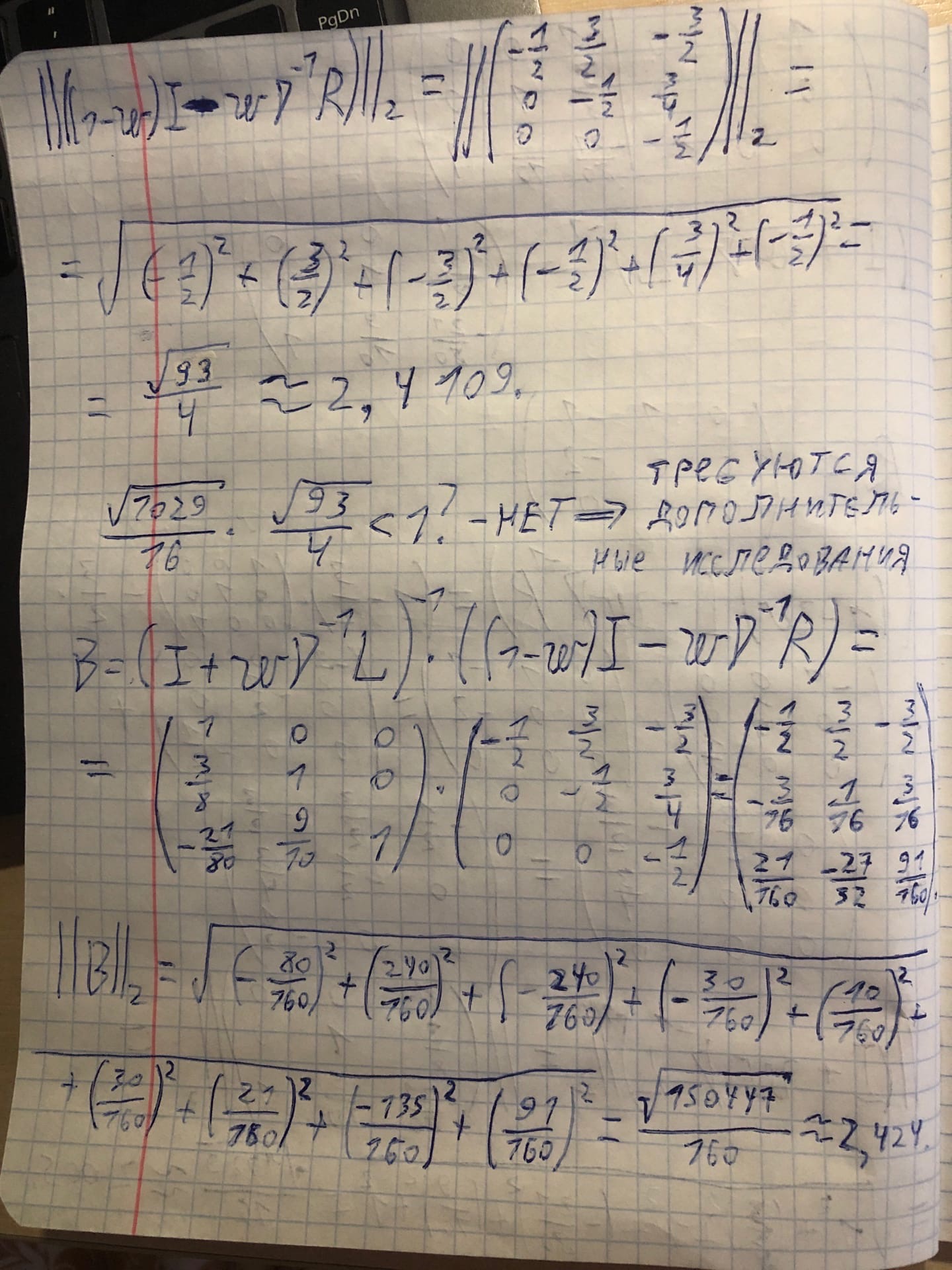


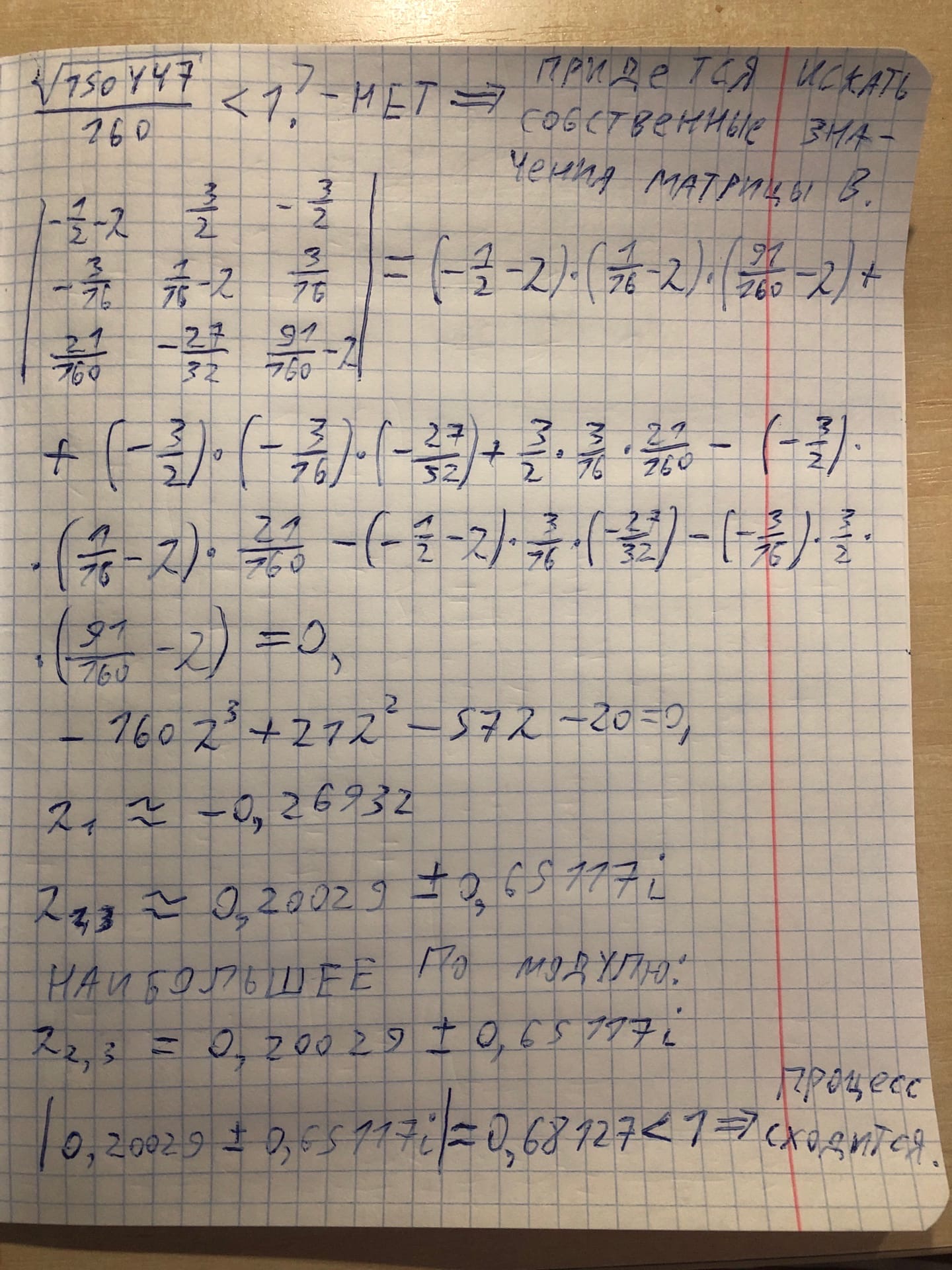
P.S. Для решения того многочлена 3-ей степени я использовал сервис WolframAplha ([https://www.wolframalpha.com/input/?i=%28-3%2F2-x%29\*%281%2F16-x%29\*%2817%2F32-x%29%2B%285%2F2%29\*%28-5%2F16%29\*%283%2F32%29%2B%28-5%2F2%29\*%28-15%2F16%29\*%28-77%2F32%29-%28-5%2F2%29\*%281%2F16-x%29\*%283%2F32%29-%28-3%2F2-x%29\*%28%28-5%2F16%29\*%28-77%2F32%29%29-%28-15%2F16%29\*%285%2F2%29\*%2817%2F32-x%29+%3D+0](https://www.wolframalpha.com/input/?i=%28-3%2F2-x%29*%281%2F16-x%29*%2817%2F32-x%29%2B%285%2F2%29*%28-5%2F16%29*%283%2F32%29%2B%28-5%2F2%29*%28-15%2F16%29*%28-77%2F32%29-%28-5%2F2%29*%281%2F16-x%29*%283%2F32%29-%28-3%2F2-x%29*%28%28-5%2F16%29*%28-77%2F32%29%29-%28-15%2F16%29*%285%2F2%29*%2817%2F32-x%29+%3D+0))

Доказательство сходимости процесса при w1 = 1.5:





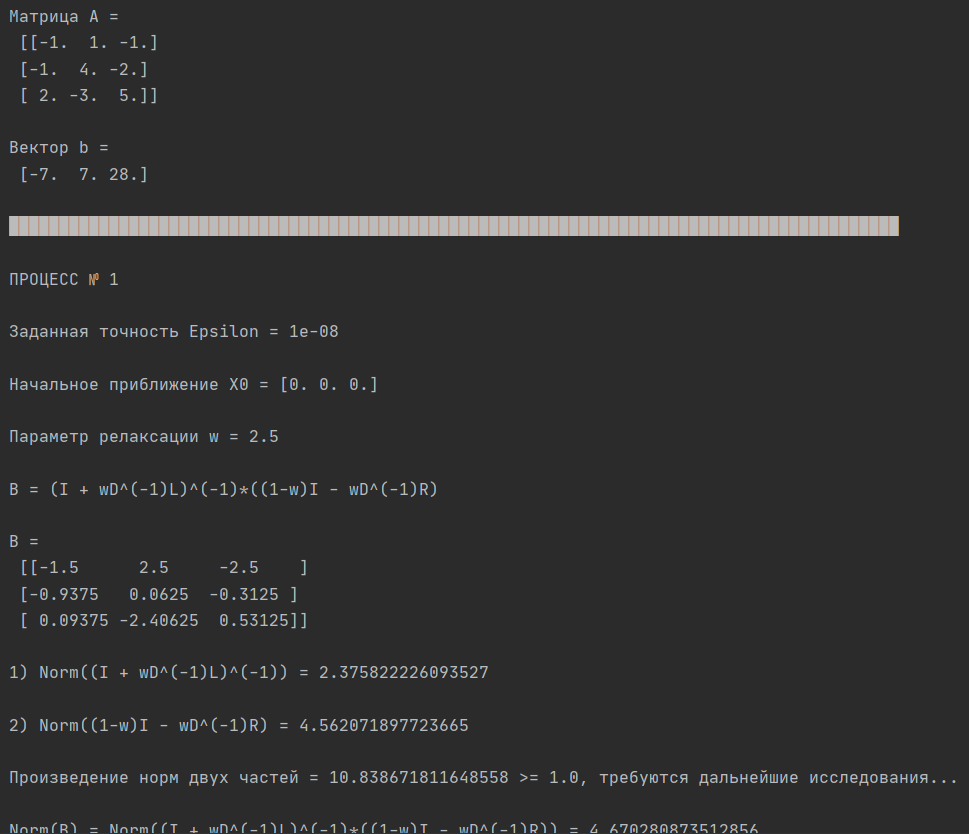


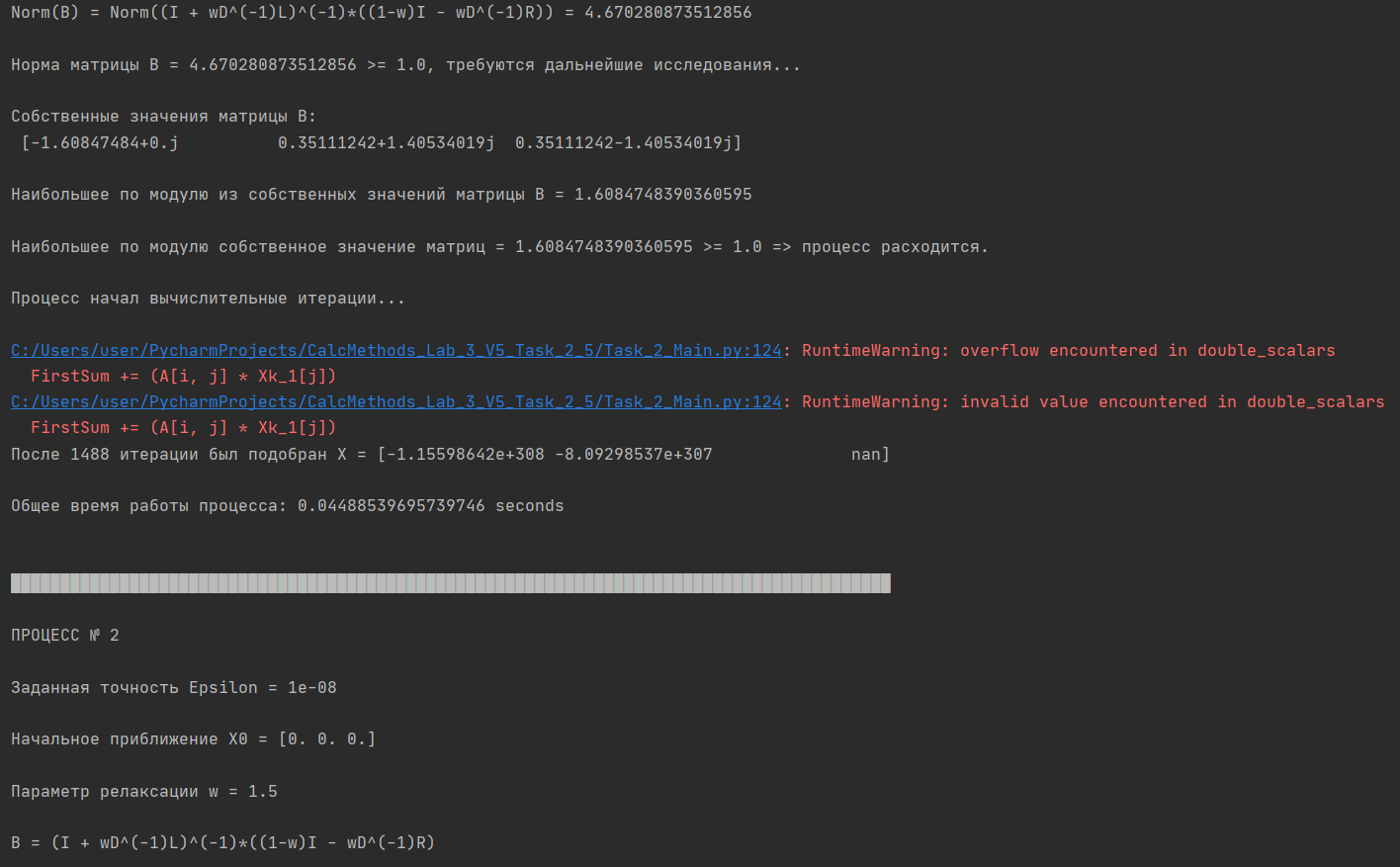


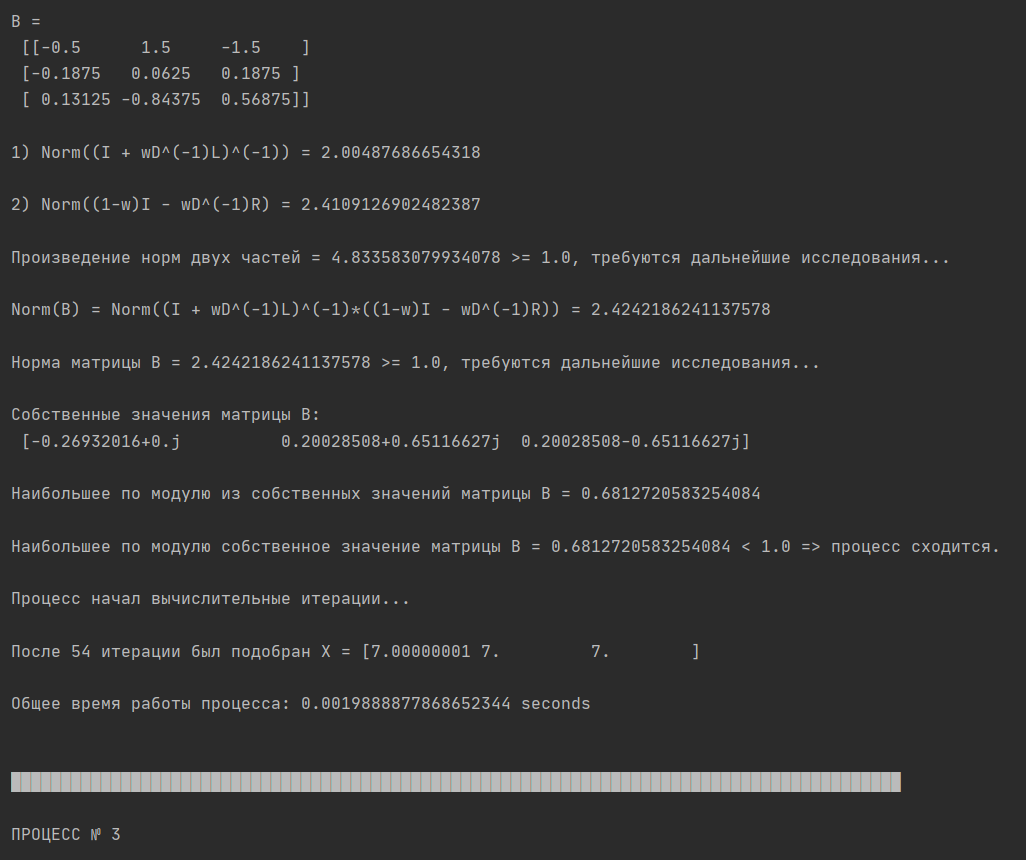
P.S. Для решения того многочлена 3-ей степени я снова использовал сервис WolframAplha ([https://www.wolframalpha.com/input/?i=%28-1%2F2-x%29\*%281%2F16-x%29\*%2891%2F160-x%29%2B%28-3%2F2%29\*%28-3%2F16%29\*%28-27%2F32%29%2B3%2F2\*3%2F16\*21%2F160-%28-3%2F2%29\*%281%2F16-x%29\*21%2F160-%28-1%2F2-x%29\*3%2F16\*%28-27%2F32%29-%28-3%2F16%29\*3%2F2\*%2891%2F160-x%29+%3D+0](https://www.wolframalpha.com/input/?i=%28-1%2F2-x%29*%281%2F16-x%29*%2891%2F160-x%29%2B%28-3%2F2%29*%28-3%2F16%29*%28-27%2F32%29%2B3%2F2*3%2F16*21%2F160-%28-3%2F2%29*%281%2F16-x%29*21%2F160-%28-1%2F2-x%29*3%2F16*%28-27%2F32%29-%28-3%2F16%29*3%2F2*%2891%2F160-x%29+%3D+0))

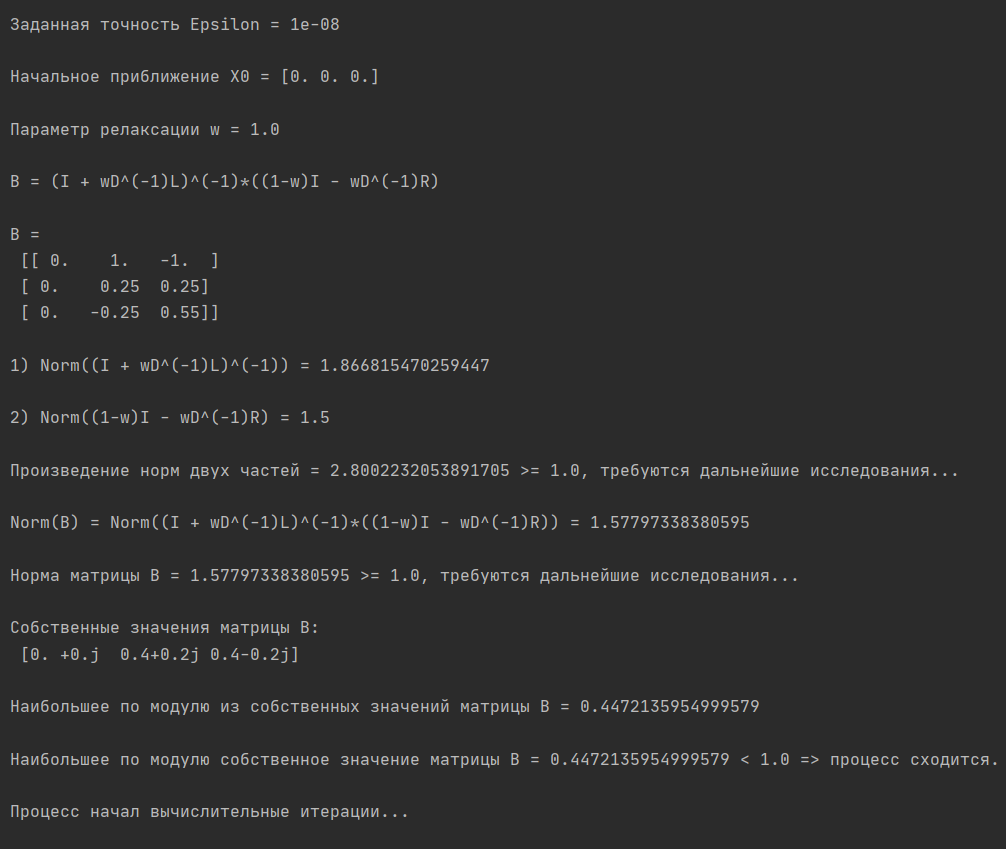
В качестве еще трёх значений параметра релаксации (w2, w3 и w5) я взял числа из диапазона от нуля до двух, а именно: w2 = 1.0, w3 = 0.5, w5 = 0.1. Как я выяснил позже, при каждом из этих трёх параметров релаксации процесс сходился, хотя и медленнее (ему требовалось больше итераций для достижения нужной точности).

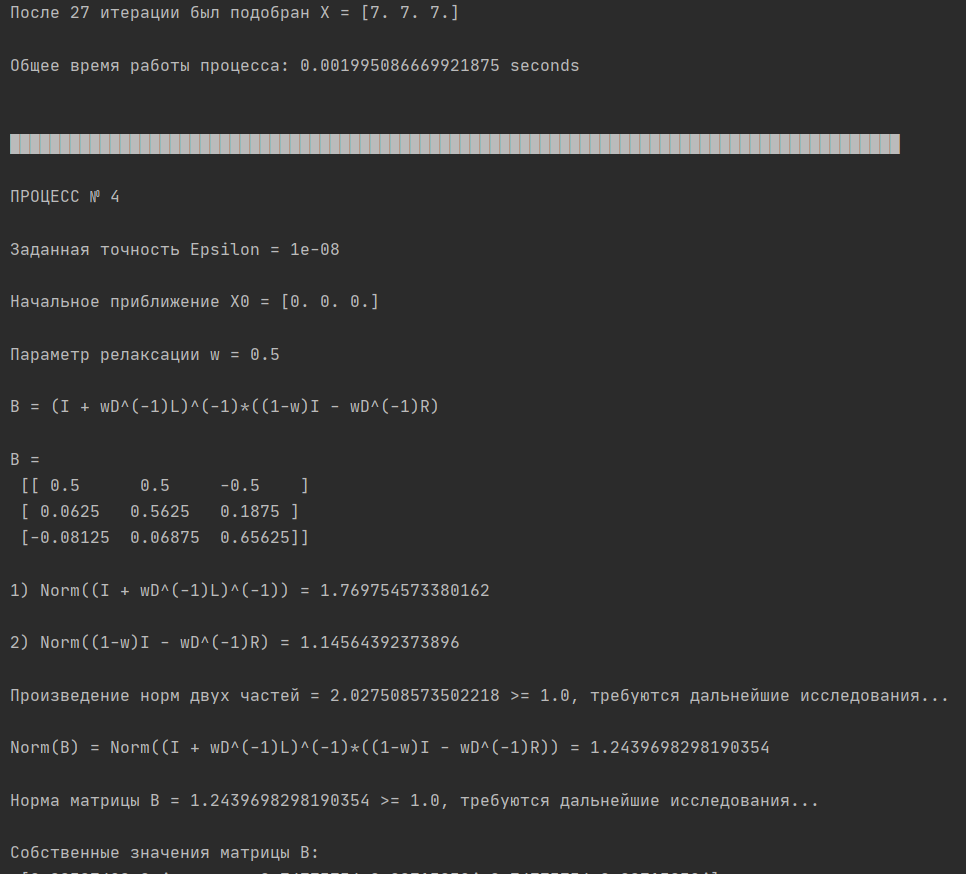
В конце концов программа закончила свои расчёты для каждого параметра релаксации (в случае с w0 она остановила расчёты перед переполнением стека) вывела следующее:

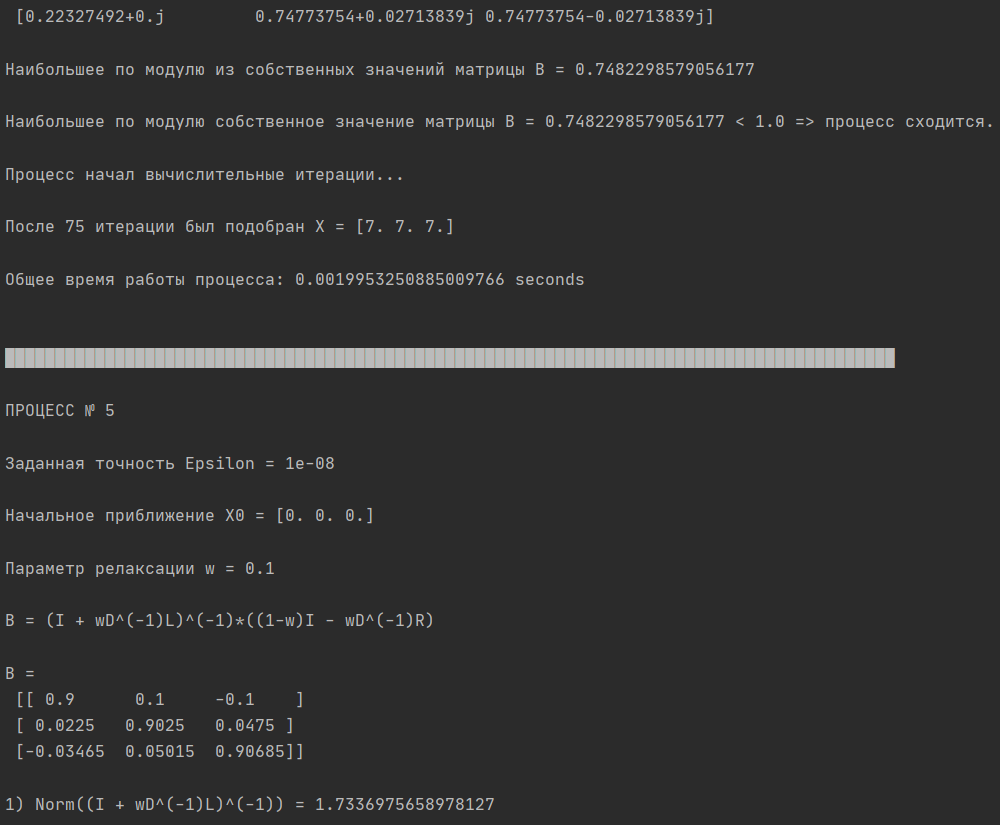


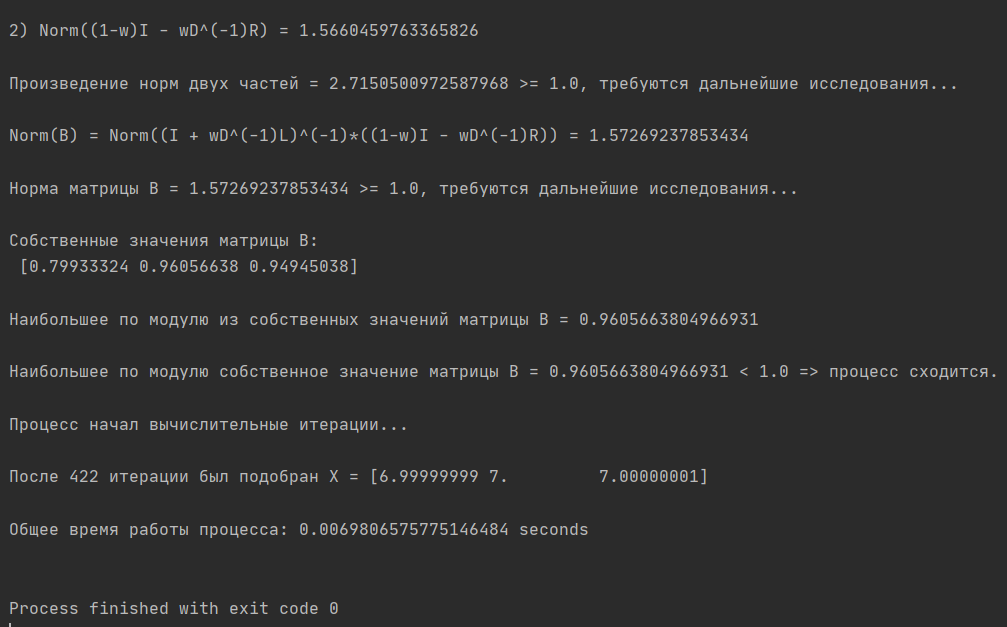






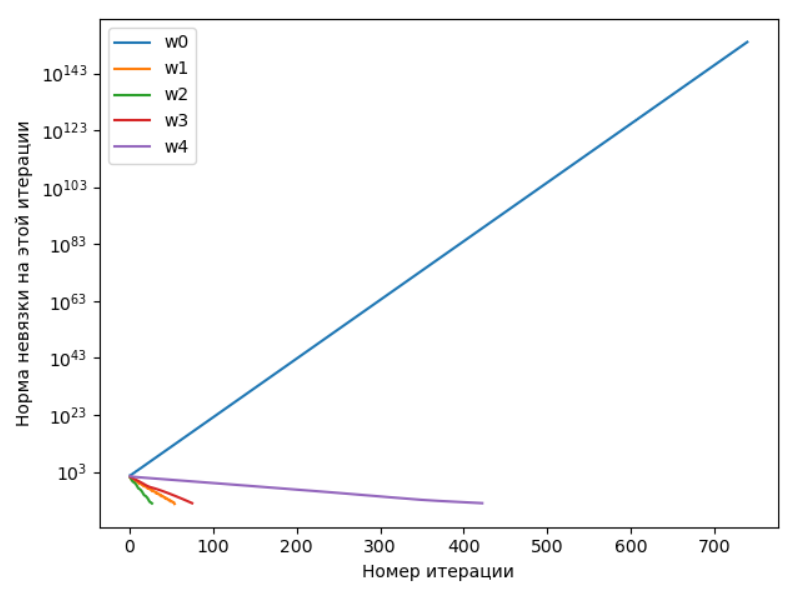






Мы можем наблюдать, что наши теоретические расчёты для параметров релаксации w0 и w1 полностью подтвердились (в том числе и промежуточные расчёты). Также мы можем наблюдать некоторые погрешности в ответах. Так, например, в 5-ом процессе при w4 = 0.1 вектор x равен {6.99999999, 7.0, 7.0} вместо истинных {7.0, 7.0, 7.0}. Однако самое главное, что требуемая точность, которую мы задали (а я задал Epsilon = 10-8) выполняется, значит программа работает корректно.

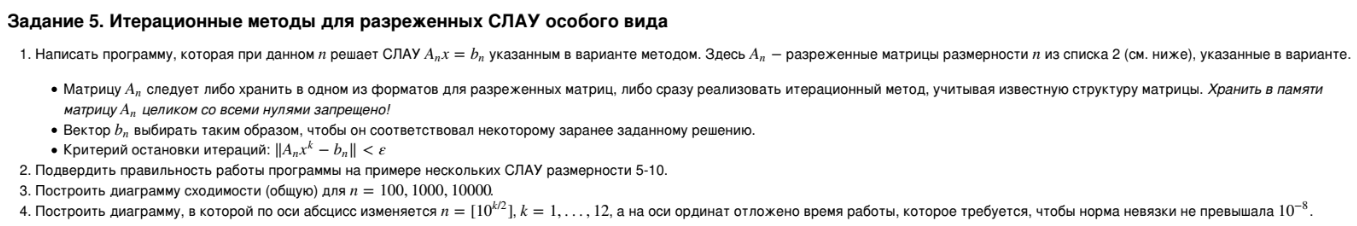
Программа также вывела следующую логарифмическую диаграмму сходимости:



Мы можем наблюдать, что при w0 = 2.5 (при котором итерационный процесс расходится) с каждой итерацией норма невязки становилась все больше и больше, что не удивительно. А что касается остальных параметров релаксации (w1, w2, w3, w4), то их итерационные процессы вполне себе сошлись, при чём можно заметить, что быстрее всего сошёлся процесс w2 = 0.5. Значится, «самый быстрый» параметр сходимости около w\* = 0.5 для нашей матрицы.

Код этой программе будет в файле Task\_2\_Main.py, а картинка с графиком в файле Task\_2\_Diagram.png.

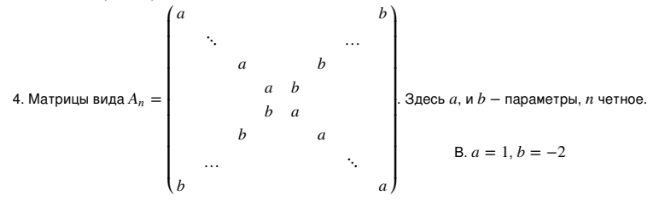
**Задание 5:**

****

**Вариант метода:**

****

**Вид матриц:**

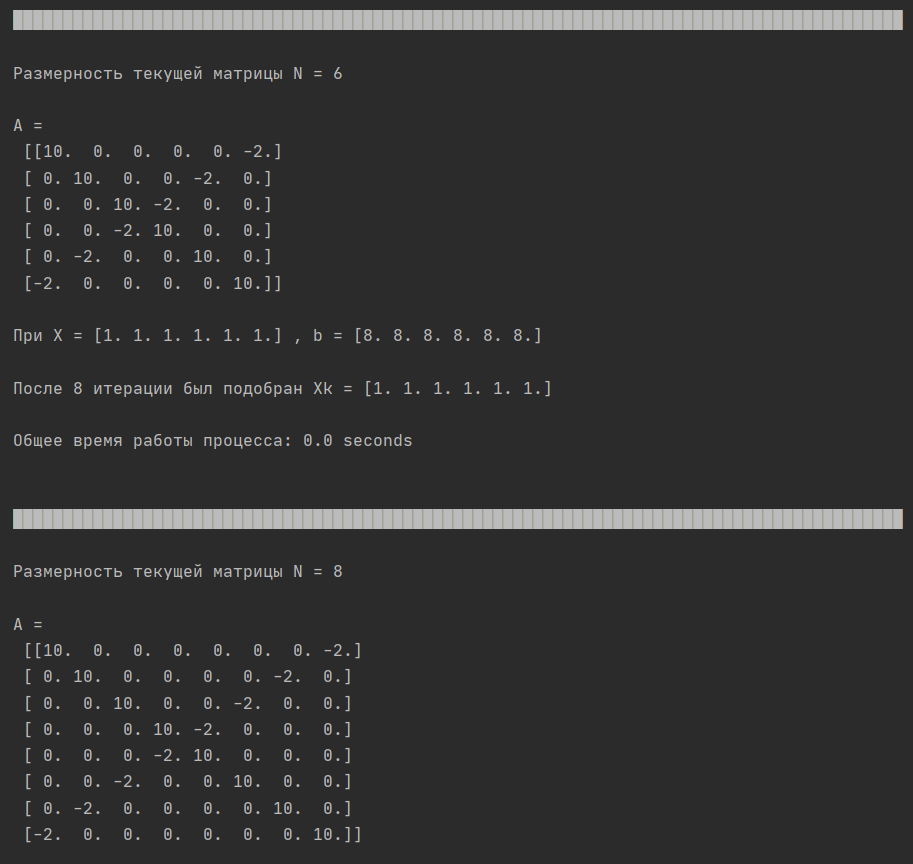
****

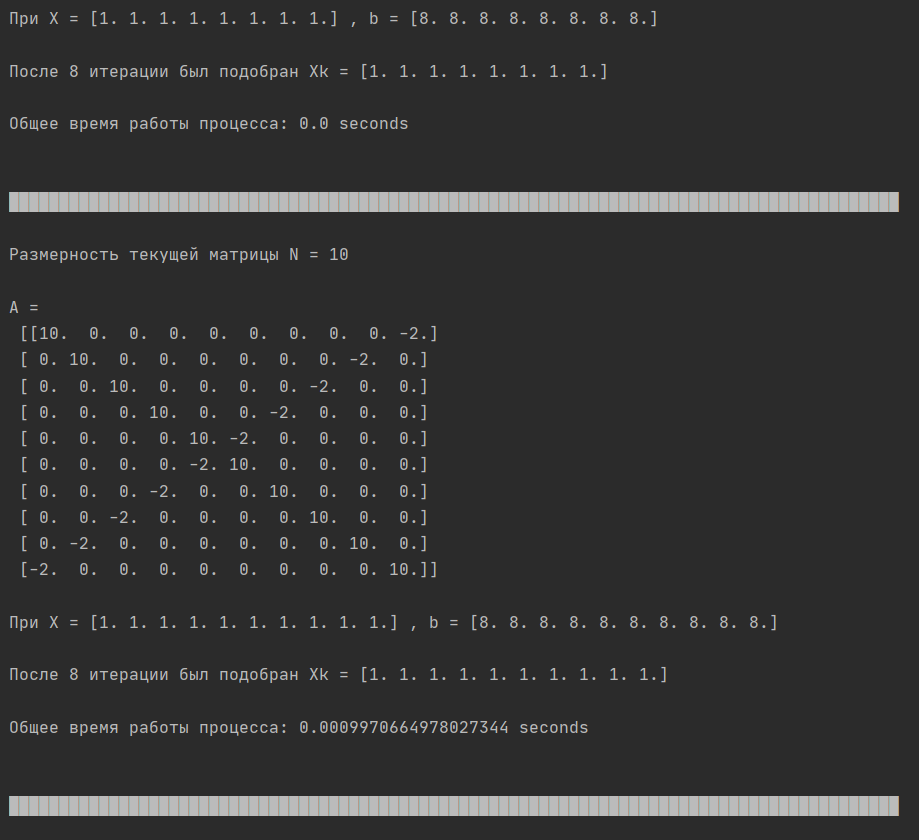
Итак, я написал программу на языке Python, которая при заданном N решает СЛАУ Anx = bn методом Гаусса-Зейделя. Для этого мне был предложен вариант матрицы, указанный на картинке выше с параметрами a = 1, b = -2. Однако после долгих исследований я пришел к выводу, что метод Гаусса-Зейделя будет расходится для матриц с такими параметрами, поэтому я взял параметр a равным 10. Таким образом, у меня были параметры a = 1, b = -2.

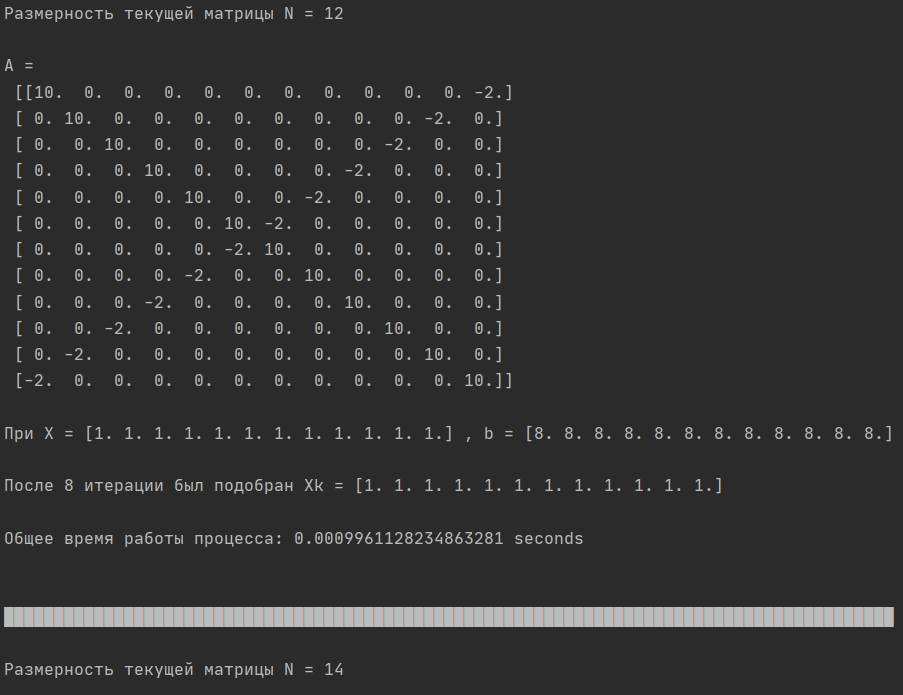
Прилагаю исходный код программы:

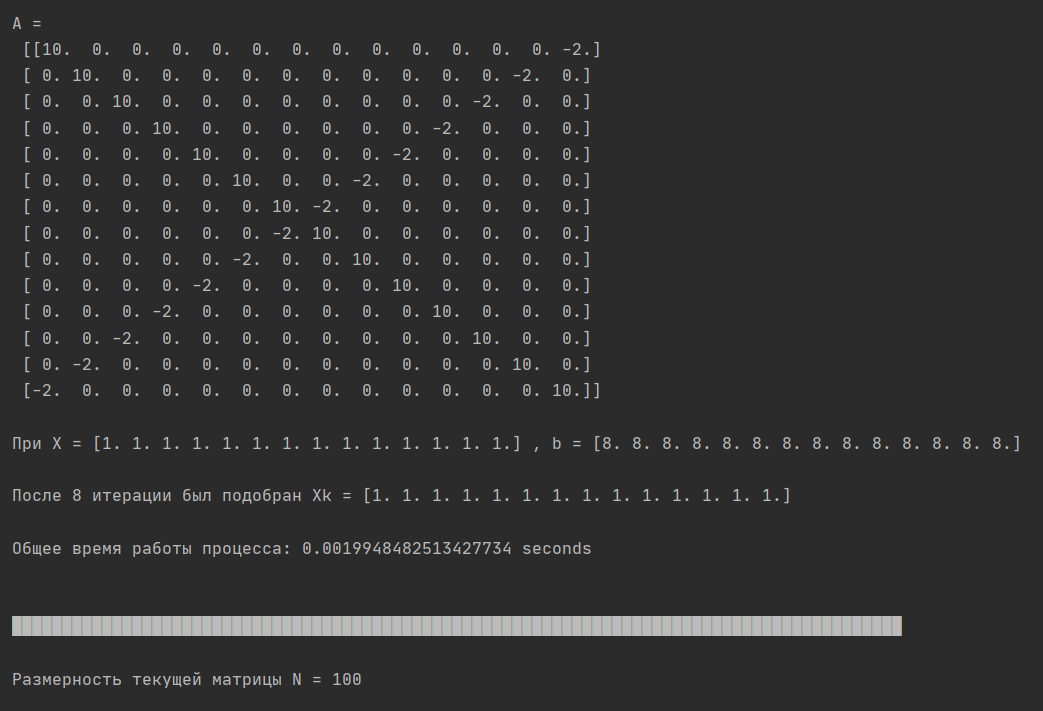
import matplotlib.pyplot as plt  
import numpy as np  
import time  
  
# Лабораторная №3, Вариант №5, Задание №5-2, Матрица №4В  
# Задание №2. Метод релаксации 1  
# 1. Написать программу, которая при данном n решает СЛАУ Anx = bn указанным в варианте методом. Здесь An - разреженные  
# матрицы размерности n из списка 2 (см. ниже), указанные в варианте.  
# - Матрицу An следует либо хранить в одном из форматов для разреженных матриц, либо сразу реализовать итерационный  
# метод, учитывая известную структуру матрицы. Хранить в памяти матрицу An целиком со всеми нулями запрещено!  
# - Вектор bn выбирать таким образом, чтобы он соответствовал некоторому заранее заданному решению  
# - Критерий остановки итераций: ||AnXk - bn|| < Epsilon  
# 2. Подтвердить правильность работы программы на примере нескольких СЛАУ размерности 5-10.  
# 3. Построить диаграмму сходимости (общую) для n = 100, 1000, 10000  
# 4. Построить диаграмму, в которой по оси абсцисс изменяется n = [10\_k/2], k = 1, ..., 12, а на оси оридант отложено  
# время работы, которое требуется, чтобы норма невязки не превышала 10^(-8).  
  
# 2. МЕТОД ГАУССА-ЗЕЙДЕЛЯ  
  
# Требуемая точность (для итераций)  
Epsilon = 0.00000001  
  
# Параметры для наших разреженных матриц  
MatrixParameter\_a = 10 # При параметре a = 1 из условия сходимость не наблюдалась  
MatrixParameter\_b = -2  
  
# ФункцияЮ генерирующая нашу матрицу по заданной размерности (a и b - параметры)  
def GenerateSpecificMatrix(N, a, b):  
 NeededMatrix = np.zeros((N, N))  
 for i in range(N):  
 NeededMatrix[i][i] = a  
 NeededMatrix[N - i - 1][i] = b  
 return NeededMatrix  
  
# Посчитать норму невязки || A\*X = b || --> min  
def ResidualRate(A, X):  
 AX = np.dot(A, X) # A\*X  
 AX\_b = AX - b # A\*X - b  
 return np.linalg.norm(AX\_b) # || A\*X - b ||  
  
# Решение СЛАУ методом Гаусса-Зейделя  
def GaussSeidel(A, b, ResRateArr):  
 StartTime = time.time()  
 N = len(A) # Запоминаем размер текущей матрицы A  
 Xk = np.zeros(N) # Текущий вектор-ответ  
 CurrResidualRate = 1 # Текущая невязка  
 IterationsAmount = 0 # Количество итераций  
 while CurrResidualRate > Epsilon: # До тех пор, пока невязка не станет меньше точности, выполняем итерации  
 IterationsAmount += 1 # На каждой итерации приплюсовываем единицу к счетчику итераций  
 Xk\_1 = np.zeros(N) # Вектор-ответ, который будет получен на следующей итерации  
 for i in range(N):  
 FirstSum = 0  
 for j in range(i):  
 FirstSum += (A[i][j] \* Xk\_1[j])  
  
 SecondSum = 0  
 for j in range(i + 1, N):  
 SecondSum += (A[i][j] \* Xk[j])  
  
 Xk\_1[i] = (-FirstSum - SecondSum + b[i]) / A[i][i]  
 Xk = np.copy(Xk\_1) # Говорим, что теперь текущий вектор-ответ - это насчитанный нами в новой итерации вектор  
 CurrResidualRate = ResidualRate(A, Xk)  
 ResRateArr.append(CurrResidualRate) # Добавляем текущую невязку в список невязок для графика  
  
 print("После", IterationsAmount, "итерации был подобран Xk =", Xk, "\n")  
  
 print("Общее время работы процесса: %s seconds" % (time.time() - StartTime), "\n")  
  
 return IterationsAmount # По завершении процесса возвращаем количество итераций, которое нам понадобилось  
  
# Проверим работу программы на матрицах размерностей: 6, 8, 10, 12, 14, 100  
ResRateArr = [] # Список списков ординат (норм невязки на разных итерациях для графика матрицы размерности i)  
IterAmount = [] # Список количеств итераций, который понадобились каждому процессу  
for i in range(6, 17, 2):  
 if i == 16:  
 i = 100 # Для размерности 100  
 print("\n█████████████████████████████████████████████████████████████████████████████████████████\n")  
 print("Размерность текущей матрицы N =", i, "\n")  
 CurrRateArr = [] # Список ординат (норм невязки на разных итерациях) для графика матрицы размерности i  
 A = GenerateSpecificMatrix(i, MatrixParameter\_a, MatrixParameter\_b)  
 print("A =\n", A, "\n")  
 X = np.ones(i) # Пуская вектором-ответом всегда будет вектор, состоящий из N единичек (так удобнее)  
 b = np.dot(A, X) # Тогда вектор b = A\*X  
 print("При X =", X, ", b =", b, "\n")  
 IterAmount.append(GaussSeidel(A, b, CurrRateArr)) # Запоминаем потребовавшееся количество итераций  
 ResRateArr.append(CurrRateArr) # Также запоминаем список невязок для текущего процесса  
  
for i in range(6):  
 LabelNum = 6 + i\*2  
 if LabelNum == 16:  
 LabelNum = 100  
 plt.semilogy(np.arange(1, IterAmount[i] + 1), ResRateArr[i], label="N = "+str(LabelNum))  
plt.xlabel("Номер итерации")  
plt.ylabel("Норма невязки на этой итерации")  
plt.legend()  
plt.show()

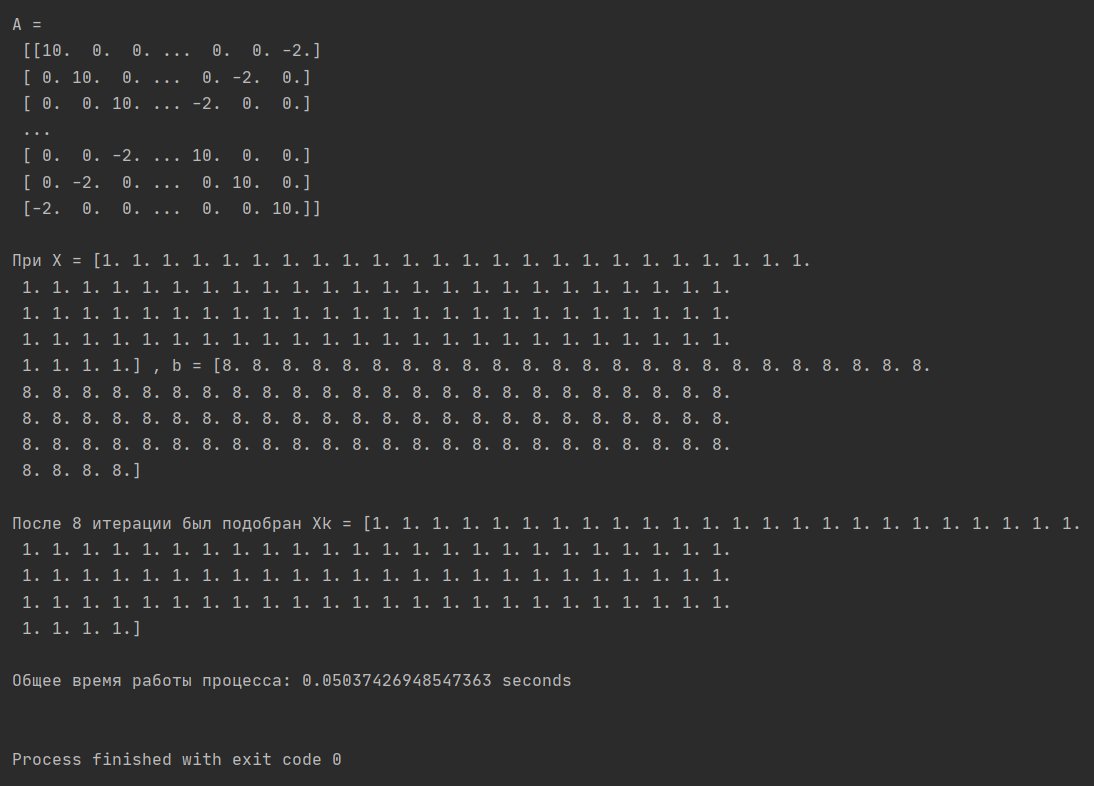
Чтобы проверить работу программы, я протестировал её на матрицах размерностей N1 = 6, N2 = 8, N3 = 10, N4 = 12, N5 = 14, N6 = 100. В качестве вектора-ответа я генерировал вектор X, состоящий из столько единичек, какой размерности была текущая матрица. Соответственно и рассчитывался вектор b. На этих матрицах программа выдала следующие результаты:





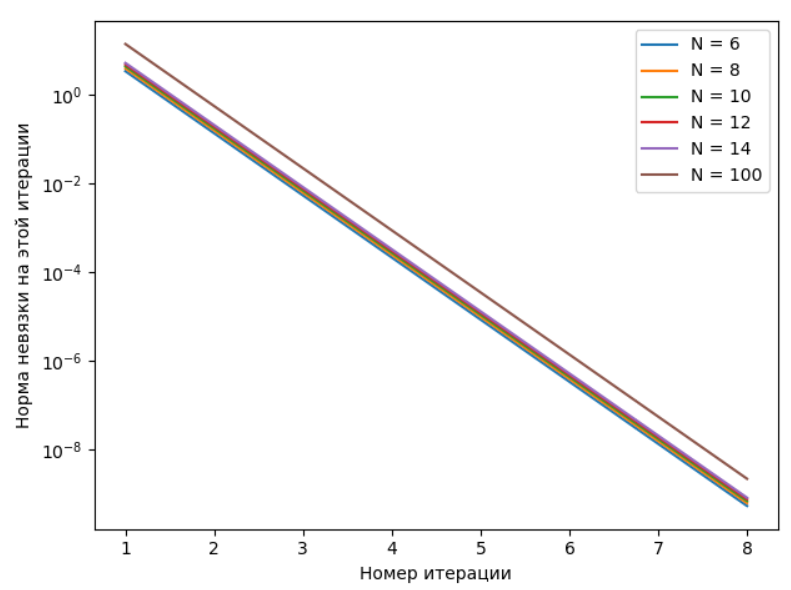






Можно наблюдать, что программа достаточно точно нашла истинное решение, при чём достаточно быстро, что говорит о корректности её работы.

Помимо всего этого, программа также выдала на экран диаграмму сходимости для всех вышеуказанных матриц:



На ней мы можем видеть, что все процессы успешно сошлись, при чём каждому потребовалось 8 итераций, чтобы достигнуть нужной точности (я выбрал точность Epsilon = 10-8).

Исходный код прикреплю в файле Task\_5\_Main.py, а картинку с диаграммой в файле Task\_5\_Diagram.png.