**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

Кафедра вычислительной математики

Жуковский Павел Сергеевич

Отчёт по лабораторной работе №4

(«Методы вычислений»)

Студента 3 курса 12 группы

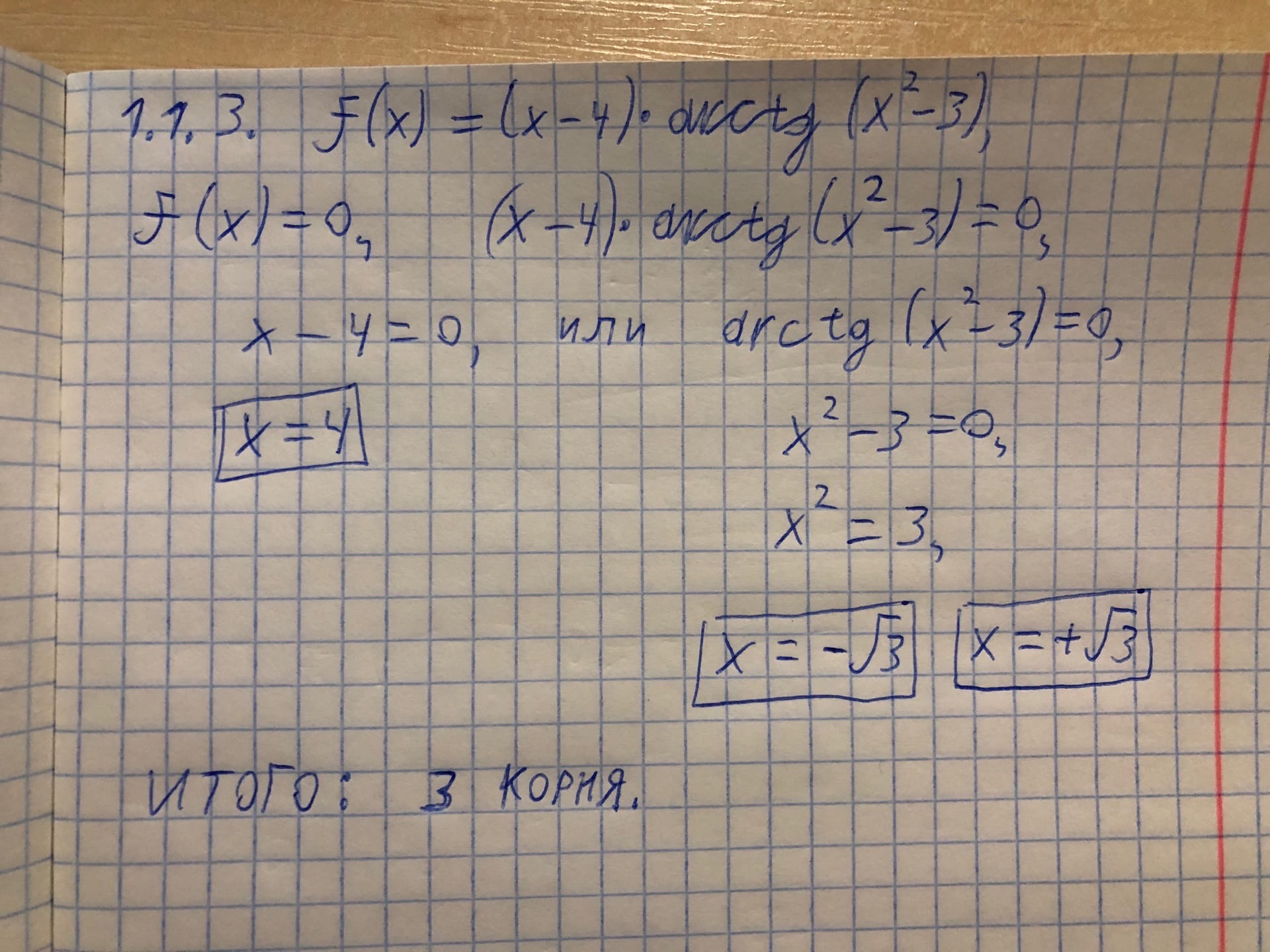
Преподаватель

Бондарь Иван Васильевич

Минск 2020

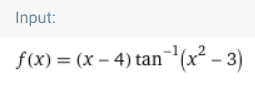
**Задание 1.1.3.** f(x) = (x – 4)\*arctg(x2 – 3), метод хорд

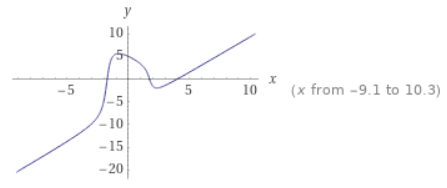
После недолгих вычислений на листике, я пришёл к выводу, что данная функция имеет 3 корня:

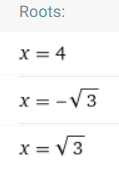


А именно, это корни: x = -, x = , x = 4.

На всякий случай я проверил себя с помощью WolframAlpha:







Теперь, когда мы знаем корни, вычисленные теоретически и точно на листике выше, перейдём к просмотру кода, который я написал с целью реализации методов Бисекции и Хорд для решения моей задачи. На всякий случай, я оставил в коде комментарии чуть ли ни к каждой строчке. Код я написал на языке программирования Python, где сделал 2 функции для двух моих методов, а потом с помощью этих функций нашёл обоими методами 3 корня на разных отрезках [a, b] (но об этом чуть позже):

from math import atan  
import numpy as np  
import matplotlib.pyplot as plt  
  
# f(x) = (x - 4)\*arctg(x^2 - 3)  
# Корни: -sqrt(3), sqrt(3), 4.  
X\_1 = -3\*\*(1/2)  
X\_2 = 3\*\*(1/2)  
X\_3 = 4  
  
# Концы отрезка [a; b] для 1-ого корня  
a1 = -3.0  
b1 = -1.5  
  
# Концы отрезка [a; b] для 2-ого корня  
a2 = 1.5  
b2 = 3.0  
  
# Концы отрезка [a; b] для 3-его корня  
a3 = 3.0  
b3 = 6.0  
  
# Точность  
Epsilon = 1e-12  
  
# Наша функция  
def Foo(X):  
 return (X - 4.0)\*atan(X\*\*2 - 3.0)  
  
# 2-ая производная от исходной функции (нужна, чтобы выбрать нужное начальное приближение в методе хорд)  
def Foo2(X):  
 return 4\*X/((3 - X\*\*2)\*\*2 + 1) - (X - 4)/(((-8)\*(3 - X\*\*2)\*(X\*\*2))/(((3 - X\*\*2)\*\*2 + 1)\*\*2) + (-2)/((3 - X\*\*2)\*\*2 + 1))  
  
# Метод Бисекции  
def BisectionMethod(f, a, b, Eps, ResRateArr, needed\_X):  
 a0 = a # Запомним, каким a было изначально  
 b0 = b # Запомним, каким b было изначально  
 Xk = 0 # Искомое решение  
 print("Расчёты итераций для Xk...")  
 Count = 0 # Счётчик итераций  
 while abs(b - a) > 2.0\*Eps:  
 Xk = (a + b) / 2.0  
 if f(Xk)\*f(a) < 0:  
 b = Xk  
 else:  
 a = Xk  
 Count += 1 # Делаем инкремент итерации на счётчике  
 nevyazka = abs(needed\_X - Xk)  
 ResRateArr.append(nevyazka) # Запоминаем невязку на каждой итерации  
 print(Count, ") Xk =", Xk, ", |X\* - Xk| =", nevyazka)  
 print("Корень функции, полученный с помощью метода бисекции на отрезке [", a0, ";", b0, "]:", Xk)  
 return Count  
  
# Метод Хорд  
def ChordMethod(f, f2, a, b, Eps, ResRateArr, needed\_X):  
 a0 = a # Запомним, каким a было изначально  
 b0 = b # Запомним, каким b было изначально  
 if np.sign(f(a)) == np.sign(f2(a)): # Если знак f(a) == f''(a), то берем X0 = a и X1 = b, иначе - берём X0 = b и X1 = a  
 X0 = a  
 X1 = b  
 else:  
 X0 = b  
 X1 = a  
 Xk = X1 - f(X1)\*((X1 - X0)/(f(X1) - f(X0))) #Xk = X2  
 Xk\_1 = f(4.0) + 1 #Xk+1, просто инициализируем чем-нибудь, отличным от 0  
 print("Расчёты итераций для Xk...")  
 Count = 0 # Счётчик итераций  
 while abs(f(Xk\_1)) >= Eps:  
 Xk\_1 = Xk - f(Xk) \* ((Xk - X0) / (f(Xk) - f(X0)))  
 Xk = Xk\_1  
 Count += 1 # Делаем инкремент итерации на счётчике  
 nevyazka = abs(needed\_X - Xk)  
 ResRateArr.append(nevyazka) # Запоминаем невязку на каждой итерации  
 print(Count, ") Xk =", Xk, ", |X\* - Xk| =", nevyazka)  
 print("Корень функции, полученный с помощью метода хорд на отрезке [", a0, ";", b0, "]:", Xk)  
 return Count # Возвращаем количество итераций, которое нам понадобилось  
  
print("\nФункция: f(x) = (x - 4)\*arctg(x^2 - 3)")  
print("Корни этой функции, расчитанные теоретически: -sqrt(3), sqrt(3), 4")  
  
print("\nПоиск первого корня методом бисекции на отрезке [", a1, ";", b1, "]")  
BM\_ResRateArr\_1 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Бисекции для 1-ого корня  
BM\_IterAmount\_1 = BisectionMethod(Foo, a1, b1, Epsilon, BM\_ResRateArr\_1, X\_1) # Делаем расчёты и запоминаем количество итераций  
BM\_IterArr\_1 = np.arange(1, BM\_IterAmount\_1 + 1) # Массив абсцисс (количества итераций) для графика Метода Бисекции для 1-ого корня  
  
print("\nПоиск второго корня методом бисекции на отрезке [", a2, ";", b2, "]")  
BM\_ResRateArr\_2 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Бисекции для 2-ого корня  
BM\_IterAmount\_2 = BisectionMethod(Foo, a2, b2, Epsilon, BM\_ResRateArr\_2, X\_2)  
BM\_IterArr\_2 = np.arange(1, BM\_IterAmount\_2 + 1) # Массив абсцисс (количества итераций) для графика Метода Бисекции для 2-ого корня  
  
print("\nПоиск третьего корня методом бисекции на отрезке [", a3, ";", b3, "]")  
BM\_ResRateArr\_3 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Бисекции для 3-ого корня  
BM\_IterAmount\_3 = BisectionMethod(Foo, a3, b3, Epsilon, BM\_ResRateArr\_3, X\_3)  
BM\_IterArr\_3 = np.arange(1, BM\_IterAmount\_3 + 1) # Массив абсцисс (количества итераций) для графика Метода Бисекции для 3-ого корня  
  
print("\nПоиск первого корня методом хорд на отрезке [", a1, ";", b1, "]")  
CM\_ResRateArr\_1 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Хорд для 1-ого корня  
CM\_IterAmount\_1 = ChordMethod(Foo, Foo2, a1, b1, Epsilon, CM\_ResRateArr\_1, X\_1)  
CM\_IterArr\_1 = np.arange(1, CM\_IterAmount\_1 + 1) # Массив абсцисс (количества итераций) для графика Метода Хорд для 1-ого корня  
  
print("\nПоиск второго корня методом хорд на отрезке [", a2, ";", b2, "]")  
CM\_ResRateArr\_2 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Хорд для 2-ого корня  
CM\_IterAmount\_2 = ChordMethod(Foo, Foo2, a2, b2, Epsilon, CM\_ResRateArr\_2, X\_2)  
CM\_IterArr\_2 = np.arange(1, CM\_IterAmount\_2 + 1) # Массив абсцисс (количества итераций) для графика Метода Хорд для 2-ого корня  
  
print("\nПоиск третьего корня методом хорд на отрезке [", a3, ";", b3, "]")  
CM\_ResRateArr\_3 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Хорд для 3-ого корня  
CM\_IterAmount\_3 = ChordMethod(Foo, Foo2, a3, b3, Epsilon, CM\_ResRateArr\_3, X\_3)  
CM\_IterArr\_3 = np.arange(1, CM\_IterAmount\_3 + 1) # Массив абсцисс (количества итераций) для графика Метода Хорд для 3-ого корня  
  
# Строим графики сходимостей  
plt.semilogy(BM\_IterArr\_1, BM\_ResRateArr\_1, label="BM1, [-3.0; -1.5]")  
plt.semilogy(BM\_IterArr\_2, BM\_ResRateArr\_2, label="BM2, [1.5; 3.0]")  
plt.semilogy(BM\_IterArr\_3, BM\_ResRateArr\_3, label="BM3, [3.0; 6.0]")  
plt.semilogy(CM\_IterArr\_1, CM\_ResRateArr\_1, label="CM1, [-3.0; -1.5]")  
plt.semilogy(CM\_IterArr\_2, CM\_ResRateArr\_2, label="CM2, [1.5; 3.0]")  
plt.semilogy(CM\_IterArr\_3, CM\_ResRateArr\_3, label="CM3, [3.0; 6.0]")  
plt.title("Диаграммы сходимости")  
plt.xlabel("Номер итерации")  
plt.ylabel("Норма невязки на этой итерации")  
plt.legend()  
plt.grid(True)  
plt.show()

Для начала отмечу, что т.к. в моей функции 3 корня, то я подобрал 3 отрезка [a; b], при которых сходятся оба метода (я постарался найти такие 3 отрезка, чтобы оба метода сходились на каждом корне в этих отрезках, что позволит нам более-менее справедливо сравнить их диаграммы сходимости, но о них позже).

Мои отрезки были следующие:

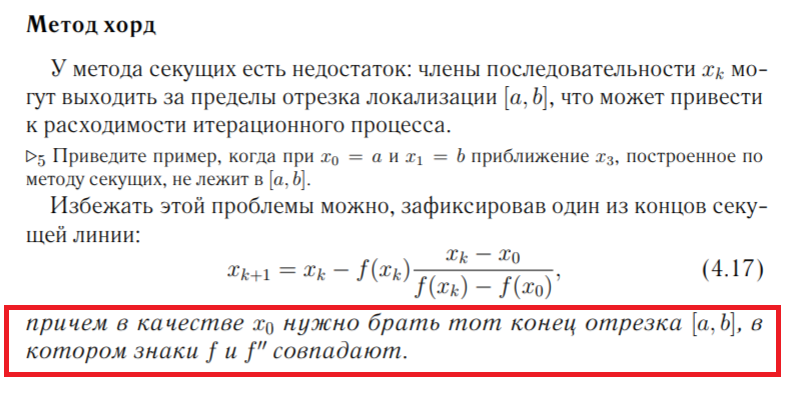
Для корня x = -, a1 = -3.0, b1 = -1.5, т.е. отрезок [-3.0; -1.5]

Для корня x = , a2 = 1.5, b2 = 3.0, т.е. отрезок [1.5; 3.0]

Для корня x = 4, a3 = 3.0, b3 = 6.0, т.е. отрезок [3.0; 6.0]

Самое главное, чтобы наши предполагаемые корни были внутри подобранных отрезков [a; b].

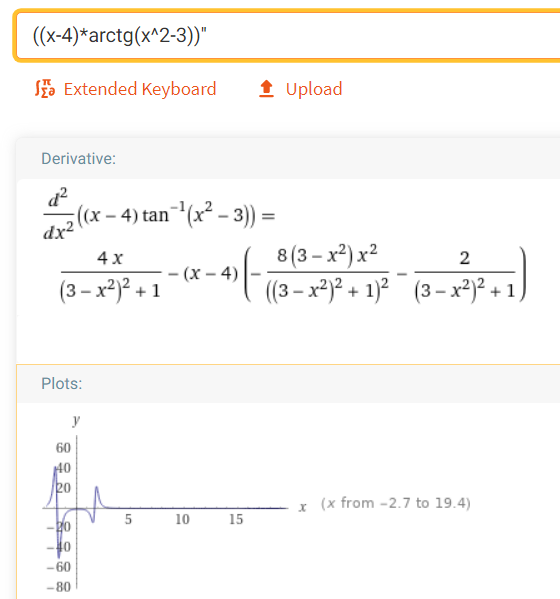
Стоит также отметить один нюанс, который касается сходимости в методе Хорд, о котором я нашёл в книжке своего лектора:



Т.е. нам важно, чтобы в качестве начального приближения был взято именно тот конец отрезка, в котором совпадают знаки функции и её 2-ой производной. Именно для этого я написал также 2-ую производную моей функцию:

# 2-ая производная от исходной функции (нужна, чтобы выбрать нужное начальное приближение в методе хорд)  
def Foo2(X):  
 return 4\*X/((3 - X\*\*2)\*\*2 + 1) - (X - 4)/(((-8)\*(3 - X\*\*2)\*(X\*\*2))/(((3 - X\*\*2)\*\*2 + 1)\*\*2) + (-2)/((3 - X\*\*2)\*\*2 + 1))

Корректность вычисления второй производной я проверил с помощью WolframAlpha:



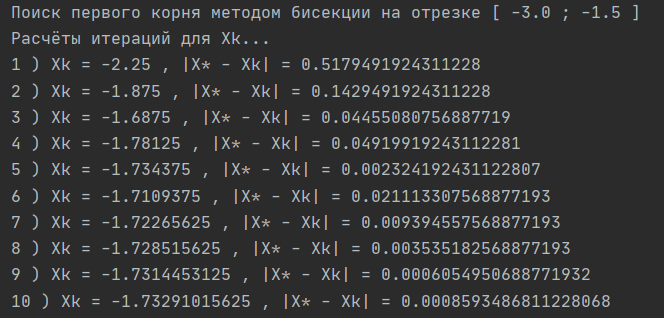
И именно для этого внутри метода Хорда я сделал вот такую проверку на знаки функции и её 2-ой производной в концах отрезка:

if np.sign(f(a)) == np.sign(f2(a)): # Если знак f(a) == f''(a), то берем X0 = a и X1 = b, иначе - берём X0 = b и X1 = a  
 X0 = a  
 X1 = b  
else:  
 X0 = b  
 X1 = a

Кстати говоря, после реализации в коде вышеуказанного нюанса некоторые мои экспериментальные отрезки, которые не сходились до этого, начали сходиться.

Что ж, поговорим о результатах вычислений.

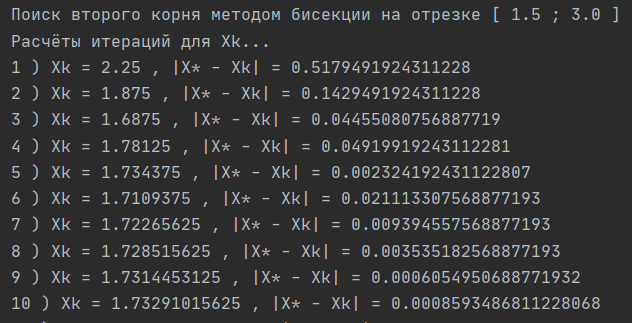
Вот, как сходился Xk в методе Бисекции на отрезке [-3.0; -1.5]:



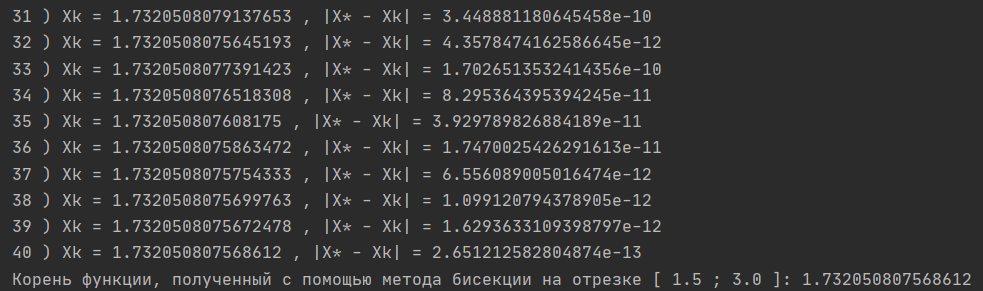
Последние итерации:



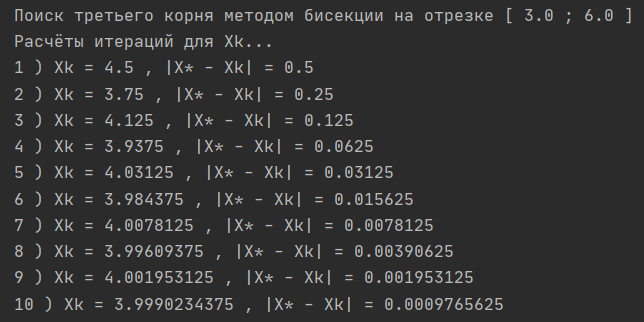
Вот, как сходился Xk в методе Бисекции на отрезке [1.5; 3.0]:



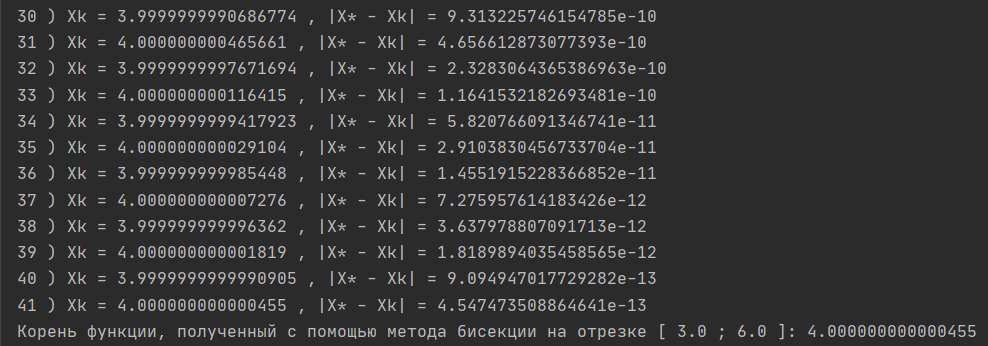
Последние итерации:



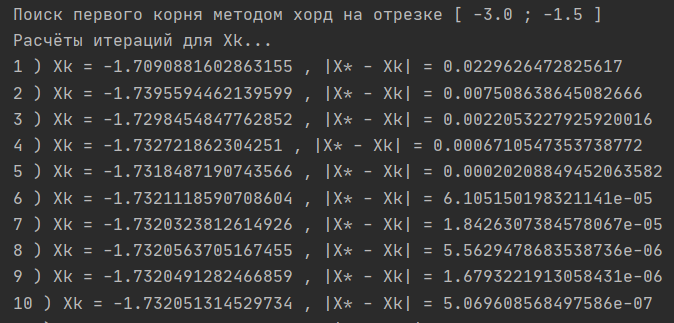
Вот, как сходился Xk в методе Бисекции на отрезке [3.0; 6.0]:



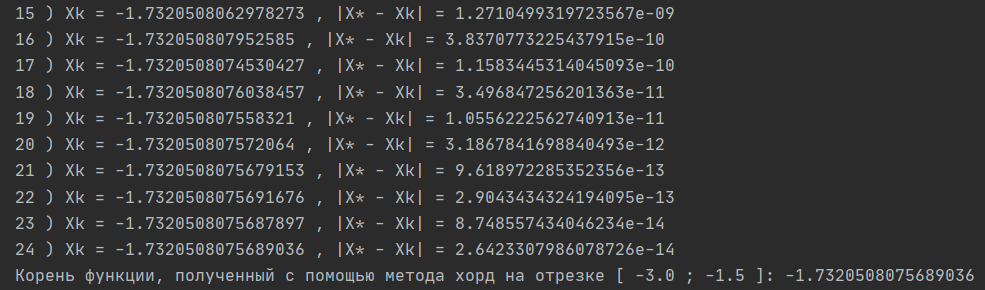
Последние итерации:



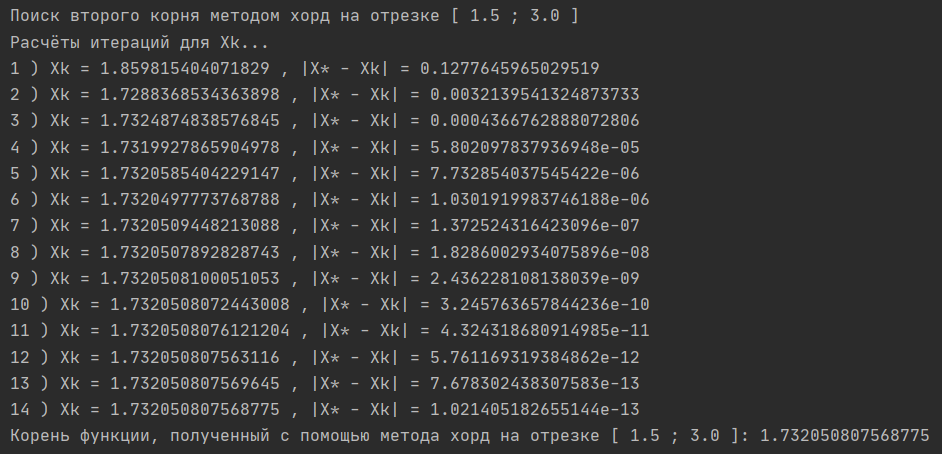
Вот, как сходился Xk в методе Хорд на отрезке [-3.0; -1.5]:



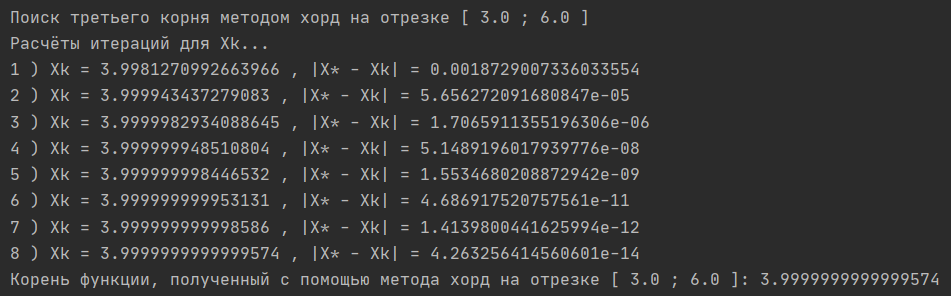
Последние итерации:



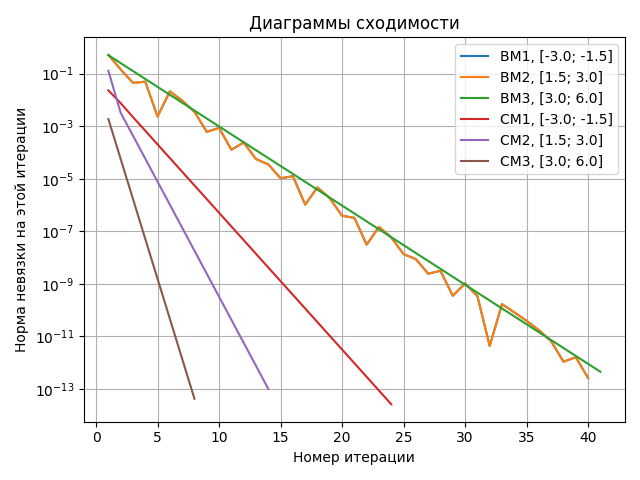
Вот, как сходился Xk в методе Хорд на отрезке [1.5; 3.0]:



Вот, как сходился Xk в методе Хорд на отрезке [3.0; 6.0]:

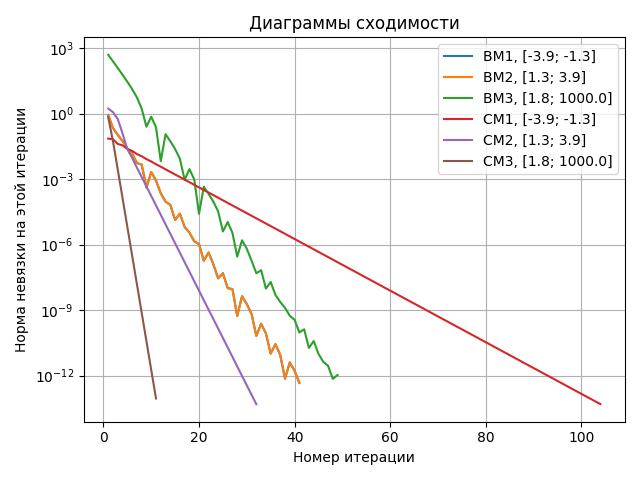


Также посмотрим на графики сходимостей, где отображена невязка каждого метода для каждого корня (на каждом отрезке) относительно текущего номера итерации (здесь BM – означает метод Бисекции, CM – метод Хорд, а цифры после них означают номер корня (отрезка), на котором этот метод выполнялся):

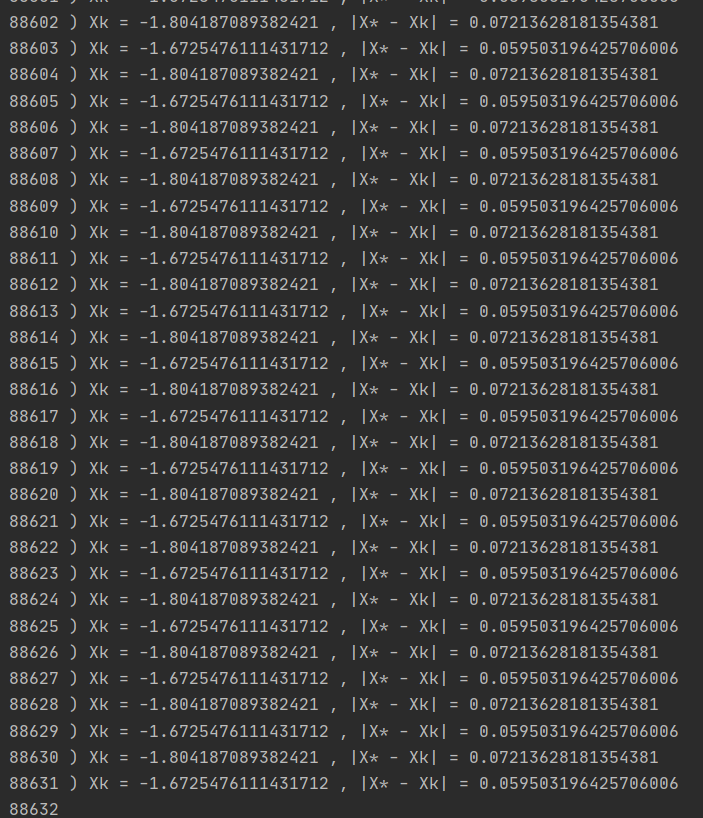


На этом графике отображены отношения невязок к номеру итерации для всех методов на всех отрезках (кстати, здесь совсем не видно голубой график для BM1, но это не удивительно, т.к. там невязки точно такие же, как и в BM2, потому что эти методы искали два противоположных корня, а именно - и ). Этот график я прикреплю в письме под именем Task\_1\_1\_3\_Графики\_Сходимостей.jpg.

Можно сказать, что оба метода нашли корни с нужной точностью (1e-12), однако на графиках видно, что метод Хорд сошёлся быстрее на заданных отрезках. Я также делал эксперименты и на других отрезках, и вот, например, были и такие ситуации:



А если брать b1 ⸦ [-1.2; 0.0] и a1 ⸦ [0.0; 1.2], то метод Хорд вообще расходится:



Так что, хоть метод Хорд и сходится немного быстрее, но для него нужно ещё подобрать хорошие приближения, иначе он либо будет сходиться медленно, либо вообще будет расходиться.

Файл task\_1\_1\_3.py с исходным кодом я прикреплю также в письме.

**Задание 2.3.** Рассмотрим систему уравнений:

Итак, я подробно изучил вышеупомянутые функции и то, как они изменяются в зависимости от параметра c. Обе эти функции являются окружностями, при чём первая с радиусом 1, а вторая – с радиусом 10. Более того, при увеличении параметра c первая окружность уходит вправо на графике (ровно по оси Ox), а вторая окружность уходит вверх на графике (ровно по оси Oy), а при уменьшении – первая окружность уходит уже влево, а вторая окружность уходит уже вниз.

Чтобы наглядно представить это, я построил эту ситуацию в программе Geogebra, где построил обе окружности по уравнениям и где добавил ползунок для параметра c, результаты моих экспериментов можно кратно изобразить на следующем gif-файле (этот файлик я также оставлю в письме с именем Наглядное\_движение\_окружностей.gif):

<https://s8.gifyu.com/images/Task_2_3_KORNI_FUNKTII_PRI_RAZNYK_c.gif>

На графиках чётко видно, что:

1) **Есть всего 3 диапазона для параметра c, при которых нету корней**: маленькая окружность слева сверху от большой, маленькая окружность внутри большой, маленькая окружность справа снизу от большой;

2) **Есть 4 значения для параметра c, при которых есть только один корень**, а именно это случаи, когда окружности касаются: маленькая окружность касается большой внешне слева сверху, маленькая окружность касается большой внутренне слева сверху, маленькая окружность касается большой внутренне справа снизу, маленькая окружность касается большой внешне справа снизу.

3) **Есть 2 диапазона для параметра c, при которых есть два корня**, а именно: когда маленькая окружность пересекается с большой слева сверху, когда маленькая окружность пересекается с большой справа снизу.

В своей лабораторной работе я решил рассмотреть по одному значению параметра c для каждого из 3-ёх вышеупомянутых случаев: 1) одно значение c = 7, при котором наша система имеет 2 корня; 2) одно значение c = , при котором наша система имеет ровно один корень; 3) одно значение c = 10, при котором наша система вообще не имеет корней. Для каждого из этих случаев я предоставлю письменные доказательства своих вычислений, а также сверю их с показаниями WolframAlpha и с результатами вывода моей программы.

Для начала покажу код моей второй программы. Её я также сделал на языке Python, а также подписал комментарии чуть ли не к каждой строчке (на всякий случай). Метод Ньютона я реализовал в отдельном методе. Код программы (будет прикреплен в письме под именем task\_2\_3.py):

from math import sqrt  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Точность  
Epsilon = 1e-12  
  
# Пусть c = 7.0, в этом случае, согласно теории, есть 2 корня  
c = 7.0 # Параметр c, при котором есть 2 корня  
X1 = (1379.0 - sqrt(19159.0))/196.0 # Первый корень (расчитан теоретически на листочке)  
X2 = (1379.0 + sqrt(19159.0))/196.0 # Второй корень (расчитан теоретически на листочке)  
  
# Наша функция, зависящая от X, после того, как мы подставили параметр c в систему, и подстановкой выразили Y через X  
def Foo1(X):  
 return 392.0\*X\*\*2 - 5516.0\*X + 19209.0  
  
# Первая производная от вышеуказанной функции  
def Foo2(X):  
 return 784.0\*X - 5516.0  
  
# Метод Ньютона  
def NyutonMethod(X0, f, f1, Eps, ResRateArr, needed\_X, solutions\_exist):  
 Xk = X0 # Искомое решние  
 Xk\_1 = Xk + 1.0 # Xk+1  
 Count = 0 # Счётчик итераций  
 nevyazka = abs(needed\_X - X0) # Невязка  
 first\_iteration = True # Флажок для первой итерации  
 print("Расчёты итераций для Xk...")  
 while abs(Xk\_1 - Xk) >= Eps:  
 if first\_iteration:  
 first\_iteration = False  
 else:  
 Xk = Xk\_1  
 Xk\_1 = Xk - f(Xk)/f1(Xk) # Xk+1 = Xk - f(Xk) / f'(Xk)  
 Count += 1 # Делаем инкремент итерации на счётчике  
 nevyazka = abs(needed\_X - Xk\_1)  
 ResRateArr.append(nevyazka) # Запоминаем невязку на каждой итерации  
 print(Count, ") Xk =", Xk\_1, ", |X\* - Xk| =", nevyazka)  
 if Count == 30 and not solutions\_exist:  
 print("За 30 итераций метод Ньютона так и не смог найти никаких корней...")  
 return Count  
 print("Корень функции, полученный с помощью метода Ньютона с начальным приближением X0 =", X0, ", равен", Xk)  
 return Count  
  
print("\nПараметр c =", c, "(один из множества случаев, когда система имеет 2 корня)")  
print("Вид функции при c =", c, ": f(x) = 392x^2 - 5516x + 19209, f'(x) = 784x - 5516")  
print("Корни этой функции, расчитанные теоретически:", X1, ",", X2)  
  
X\_0 = 4.0  
print("\nПоиск первого корня методом Ньютона с начальным приближением X0 =", X\_0)  
NM\_ResRateArr\_1 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Ньютона для 1-ого корня  
NM\_IterAmount\_1 = NyutonMethod(X\_0, Foo1, Foo2, Epsilon, NM\_ResRateArr\_1, X1, True) # Делаем расчёты и запоминаем количество итераций  
NM\_IterArr\_1 = np.arange(1, NM\_IterAmount\_1 + 1) # Массив абсцисс (количества итераций) для графика Метода Ньютона для 1-ого корня  
  
X\_0 = 10.0  
print("\nПоиск второго корня методом Ньютона с начальным приближением X0 =", X\_0)  
NM\_ResRateArr\_2 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Ньютона для 2-ого корня  
NM\_IterAmount\_2 = NyutonMethod(X\_0, Foo1, Foo2, Epsilon, NM\_ResRateArr\_2, X2, True) # Делаем расчёты и запоминаем количество итераций  
NM\_IterArr\_2 = np.arange(1, NM\_IterAmount\_2 + 1) # Массив абсцисс (количества итераций) для графика Метода Ньютона для 2-ого корня  
  
# Пусть c = 11sqrt(2)/2, в этом случае, согласно теории, есть толькой один корень (это один из четырёх таких случаев)  
c = sqrt(121.0 / 2.0) # Параметр c, при котором есть только один корень  
X1 = 5\*sqrt(2) # Первый и единственый корень (расчитан теоретически на листочке)  
  
# Наша функция, зависящая от X, после того, как мы подставили параметр c в систему, и подстановкой выразили Y через X  
def Foo3(X):  
 return X\*\*2 - 10\*sqrt(2)\*X + 50  
  
# Первая производная от вышеуказанной функции  
def Foo4(X):  
 return 2\*X - 10\*sqrt(2)  
  
print("\nПараметр c =", c, "(один из четырёх случаев, когда система имеет только один корень)")  
print("Вид функции при c =", c, ": f(x) = x^2 - 10sqrt(2) + 50, f'(x) = 2x - 10sqrt(2)")  
print("Корень этой функции, расчитанный теоретически:", X1)  
  
X\_0 = 2.0  
print("\nПоиск первого и единственного корня методом Ньютона с начальным приближением X0 =", X\_0)  
NM\_ResRateArr\_3 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Ньютона для 1-ого корня  
NM\_IterAmount\_3 = NyutonMethod(X\_0, Foo3, Foo4, Epsilon, NM\_ResRateArr\_3, X1, True) # Делаем расчёты и запоминаем количество итераций  
NM\_IterArr\_3 = np.arange(1, NM\_IterAmount\_3 + 1) # Массив абсцисс (количества итераций) для графика Метода Ньютона для 1-ого корня  
  
# Пусть c = 10, в этом случае, согласно теории, корней вообщей нету (дискриминант отрицательный, пересечений нету)  
c = 10.0 # Параметр c, при котором нет корней  
  
# Наша функция, зависящая от X, после того, как мы подставили параметр c в систему, и подстановкой выразили Y через X  
def Foo5(X):  
 return 800\*X\*\*2 - 11960\*X + 49401  
  
# Первая производная от вышеуказанной функции  
def Foo6(X):  
 return 1600\*X - 11960  
  
print("\nПараметр c =", c, "(один из множества случаев, когда система вообще не имеет корней)")  
print("Вид функции при c =", c, ": f(x) = 800x^2 - 11960x + 49401, f'(x) = 1600x - 11960")  
  
X\_0 = 5.0  
print("\nПопытки найти какие-то корни Методом Ньютона с начальным приближением X0 =", X\_0)  
NM\_ResRateArr\_4 = [] # Массив ординат (норм невязки на разных итерациях) для графика Метода Ньютона  
NM\_IterAmount\_4 = NyutonMethod(X\_0, Foo5, Foo6, Epsilon, NM\_ResRateArr\_4, X1, False) # Делаем расчёты и запоминаем количество итераций  
NM\_IterArr\_4 = np.arange(1, NM\_IterAmount\_4 + 1) # Массив абсцисс (количества итераций) для графика Метода Ньютона  
  
# Строим графики сходимостей  
plt.semilogy(NM\_IterArr\_1, NM\_ResRateArr\_1, label="NM1, c = 7.0 (1-ый корень из двух")  
plt.semilogy(NM\_IterArr\_2, NM\_ResRateArr\_2, label="NM2, c = 7.0 (2-ой корень из двух)")  
plt.semilogy(NM\_IterArr\_3, NM\_ResRateArr\_3, label="NM3, c = 11sqrt(2)/2 (единственный корень)")  
plt.semilogy(NM\_IterArr\_4, NM\_ResRateArr\_4, label="NM4, c = 10 (нет корней)")  
plt.title("Диаграммы сходимости")  
plt.xlabel("Номер итерации")  
plt.ylabel("Норма невязки на этой итерации")  
plt.legend()  
plt.grid(True)  
plt.show()

Итак, теперь рассмотрим, что же было в каждом из трёх моих случаев, сходились ли корни, были ли погрешности и как себя показали графики.

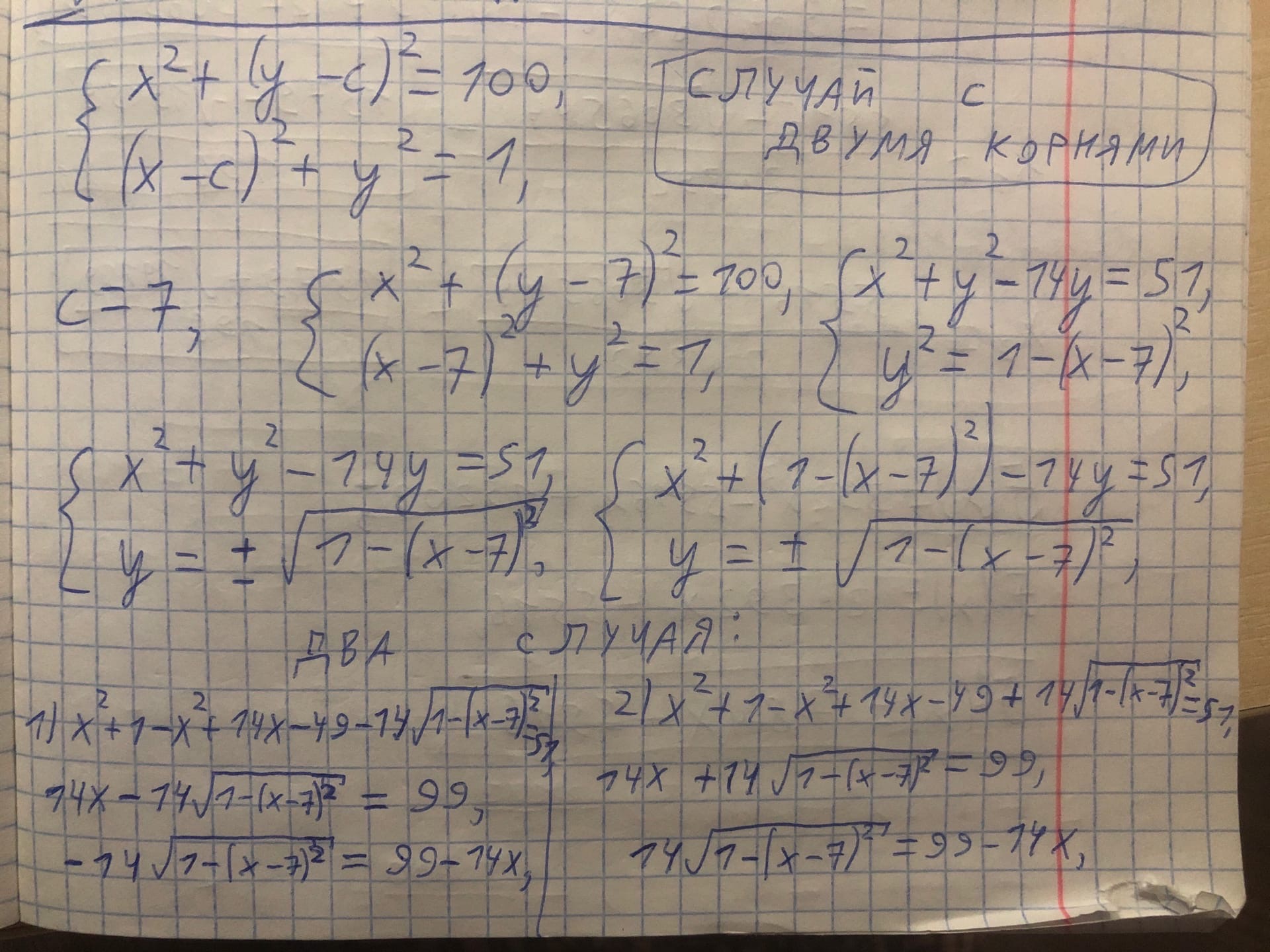
С**лучай 1 – система имеет 2 корня**

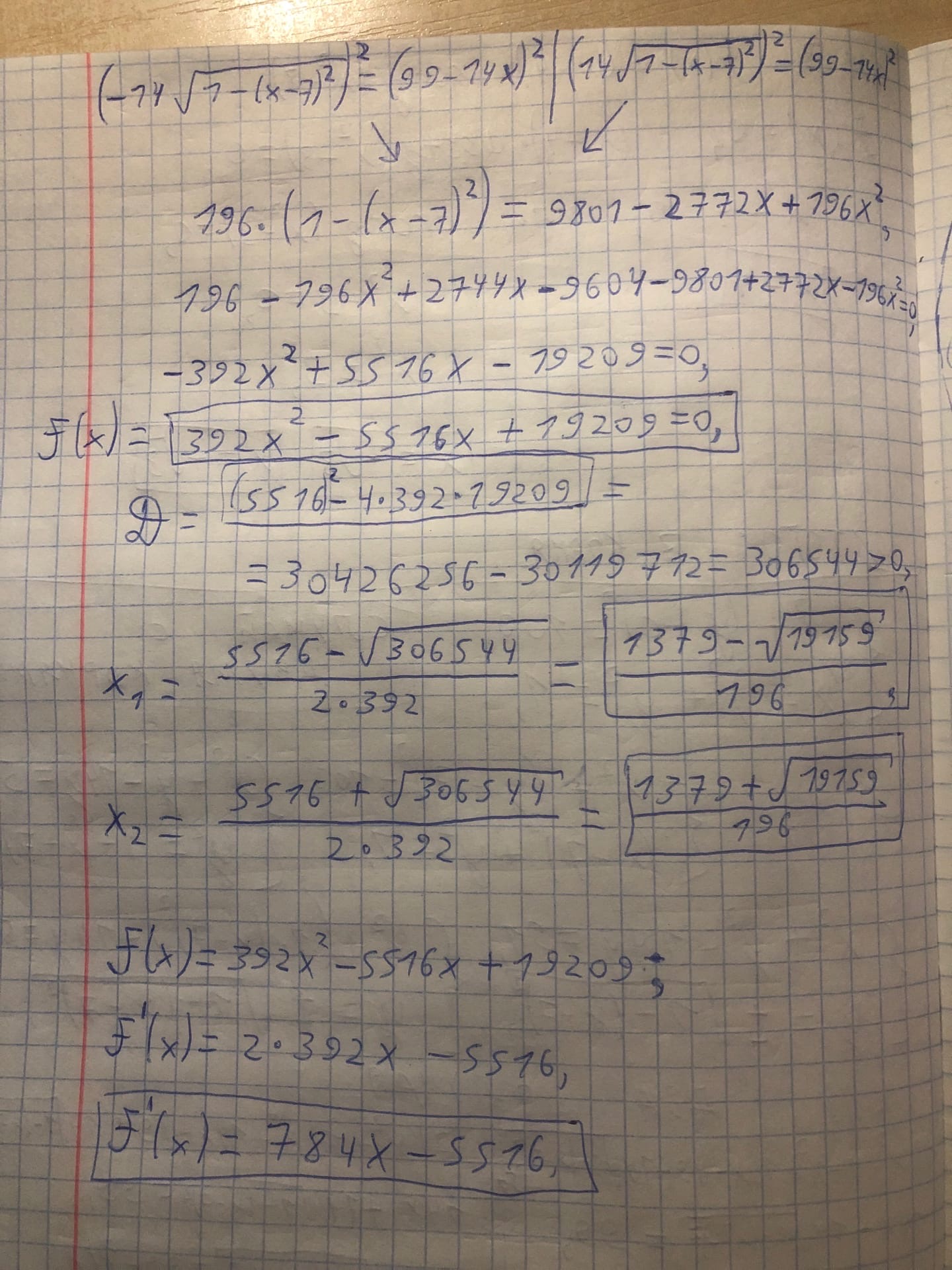
В качестве параметра c, при котором система имеет 2 корня, я просто взял какое-нибудь целое число, а именно 7, при котором графики окружностей (изображенных в программе GeoGebra) пересекаются:



Таким образом, при параметре c = 7 наша система, по идее, должна иметь 2 корня (т.к. на графике 2 пересечения окружностей). Кстати, можно было взять и какое-нибудь другое расположение окружностей с двумя пересечениями (например, слева сверху), но я для удобства взял целочисленное значение параметра c.

Чтобы доказать мою гипотезу, я письменно в тетради подставил c = 7 в оба уравнения моей системы, затем выразил во втором уравнении y через x и подставил в первое уравнение, после чего привёл его к привычному виду (к уравнению параболы) и затем, посчитав дискриминант (который оказался больше нуля), я нашёл 2 корня (хотя, они получились довольно «страшенькими»), а ещё нашёл производную моей функции (она пригодится в методе Ньютона):

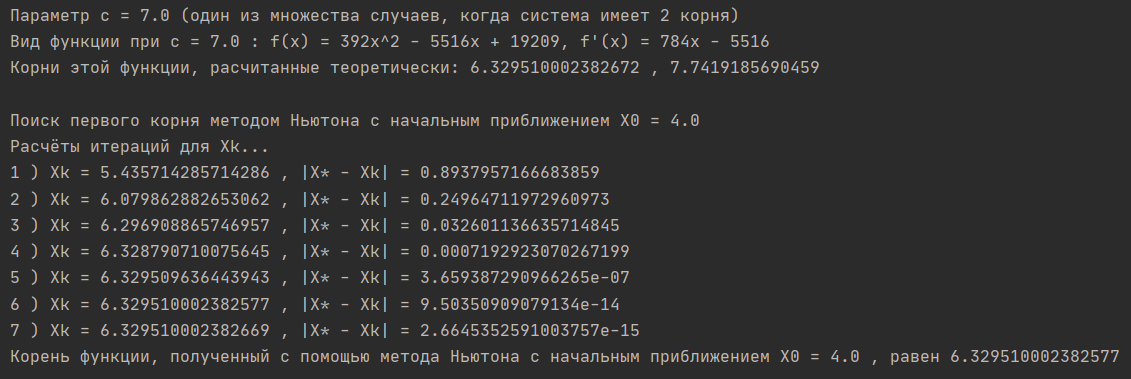




Таким образом, я теоретически нашёл корни, к которым должен сойтись мой метод Ньютона для посчитанной относительно c = 7 функции и её производной.

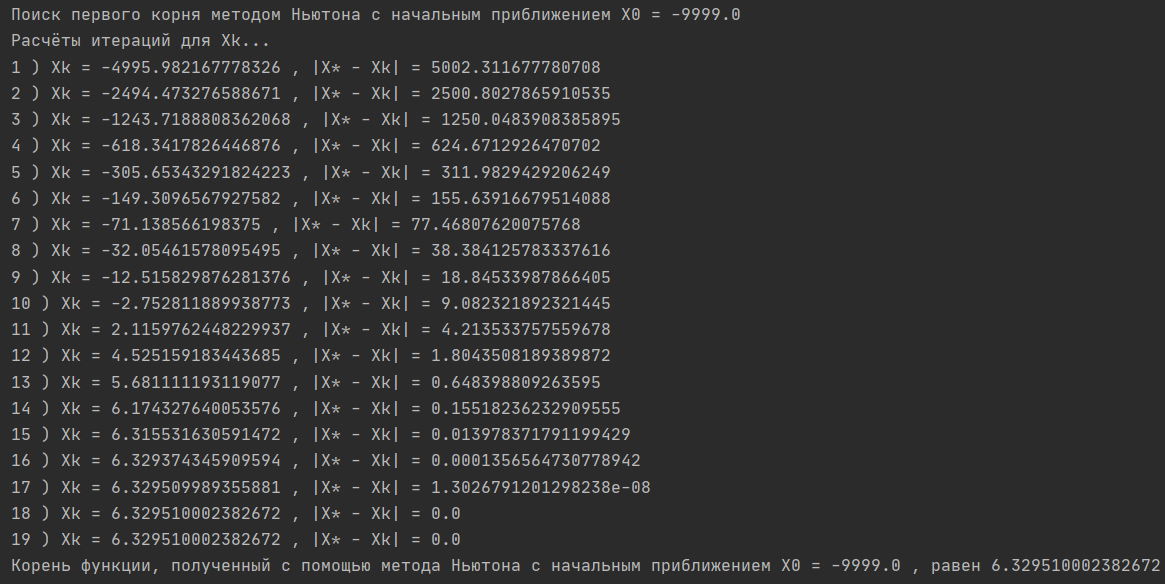
Кстати, что касается начальных приближений, то Метод Ньютона довольно гибок в этом плане. Я пробовал брать то близкие к корням числа, то далёкие, но все равно метод сходился достаточно быстро.

Итак, для начала я считал 1-ый корень (X1 = ), а в качестве приближения взял число X0 = 4:

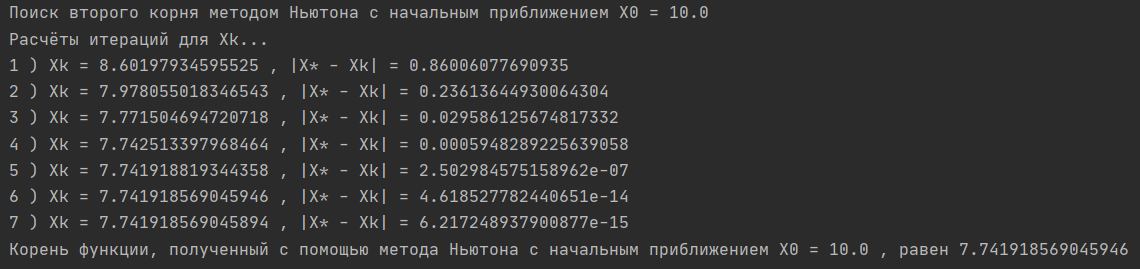


Метод на данном корне сошёлся хорошо.

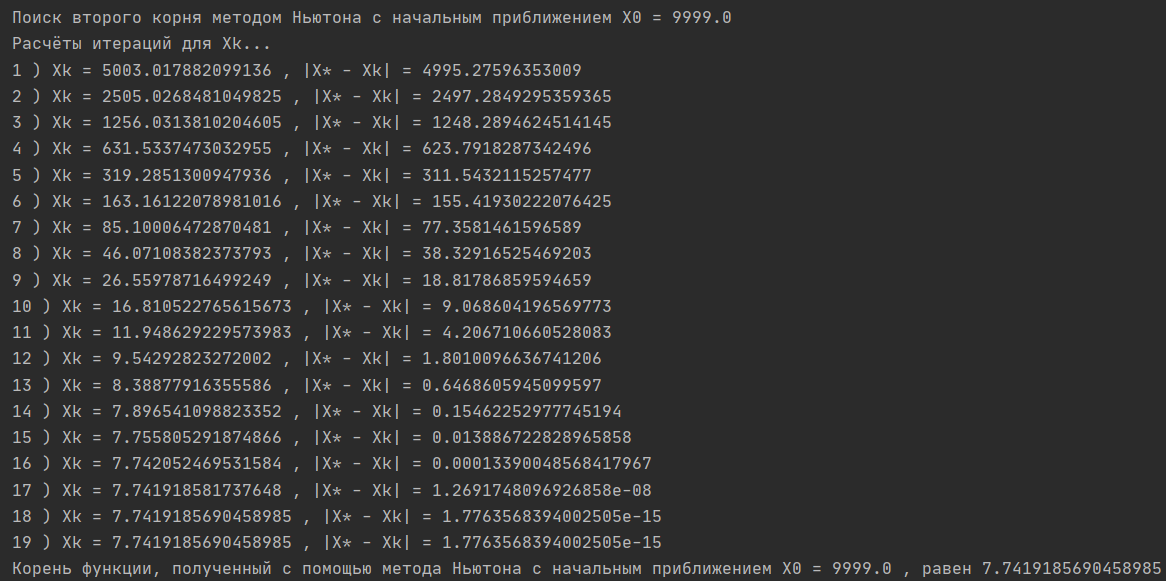
Но я также пробовал его и на более отдалённых приближениях (X0 = -9999):



Затем я также считал и 2-ой корень (X2 = ), а в качестве приближения взял число X0 = 10:



Для этого корня я также попробовал приближения, которые подальше (X0 = 9999):



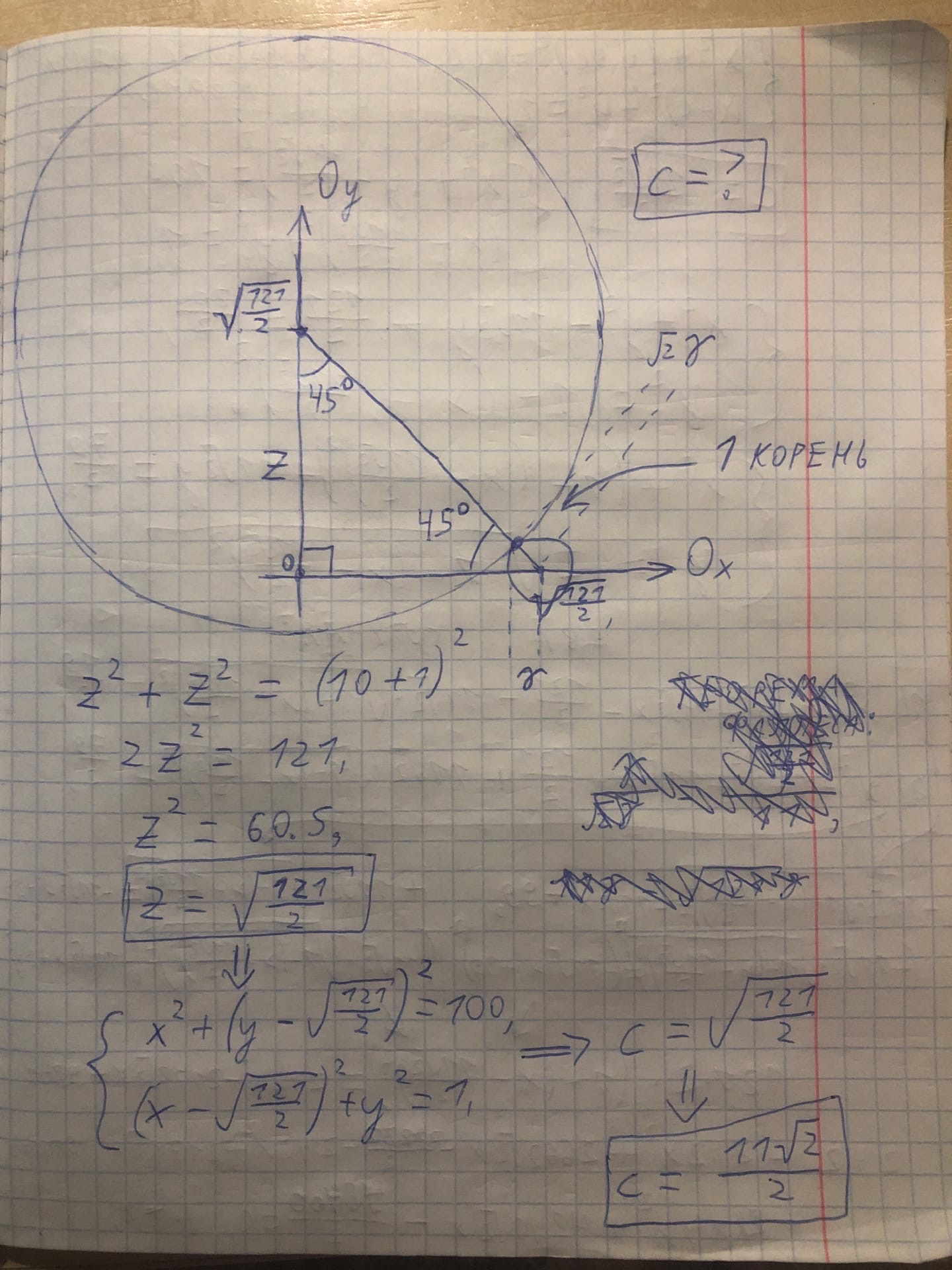
Таким образом, для случая с двумя корнями Метод Ньютона работает очень хорошо. Какое бы начальное приближение мы не взяли, оно сведётся к одному из двух корней (и насколько я понял, к тому из корней, который поближе).

С**лучай 1 – система имеет ровно 1 корень**

Вообще, этот случай возможен лишь в 4-ёх точках (лишь 4-мя способами окружности могут касаться, я надеюсь, очевидно, какими, ~~пожалуйста, не заставляйте меня это доказывать~~).

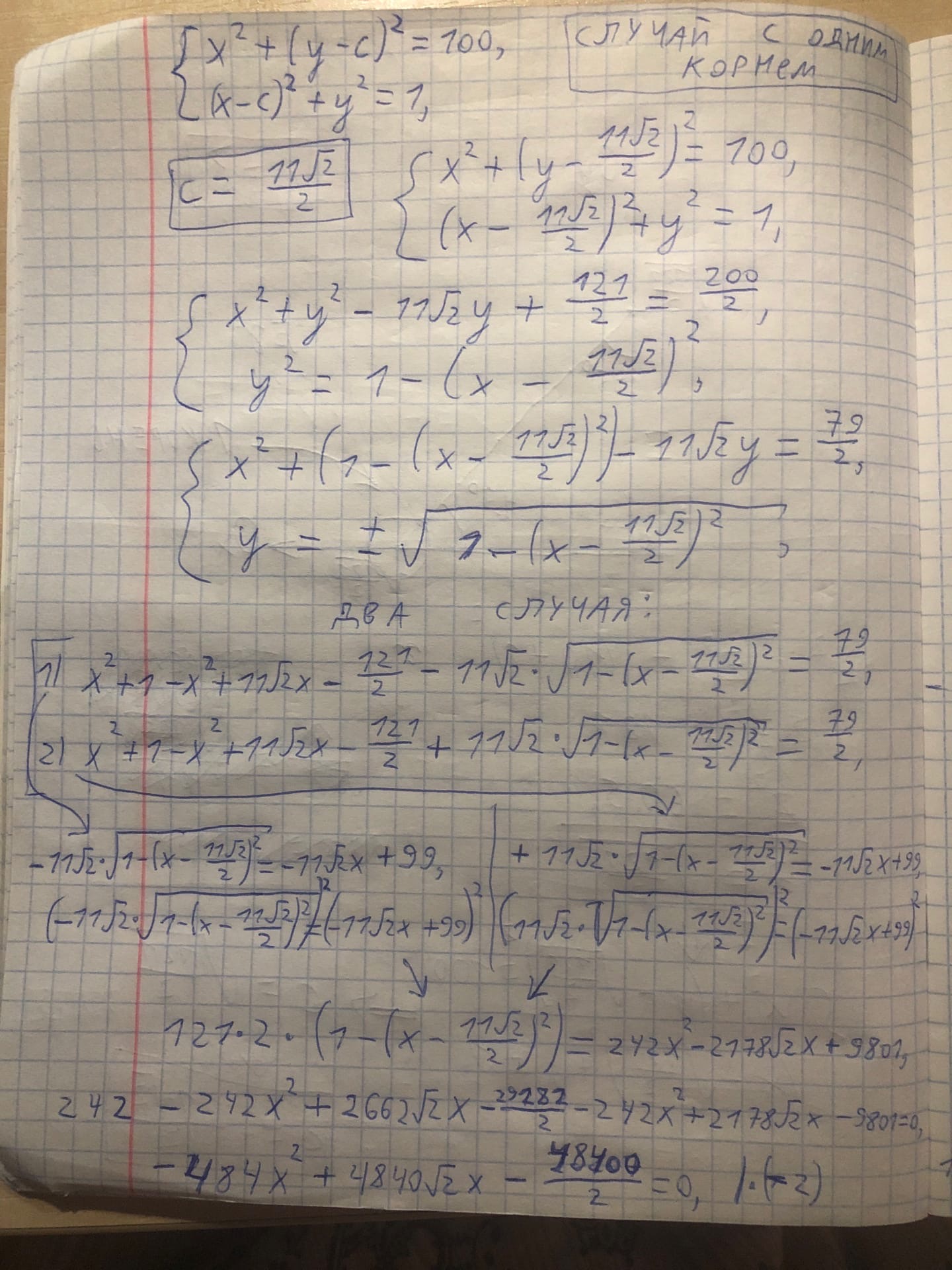
Для своего случая я взял точку, в которой окружности касаются внешним образом справа снизу.

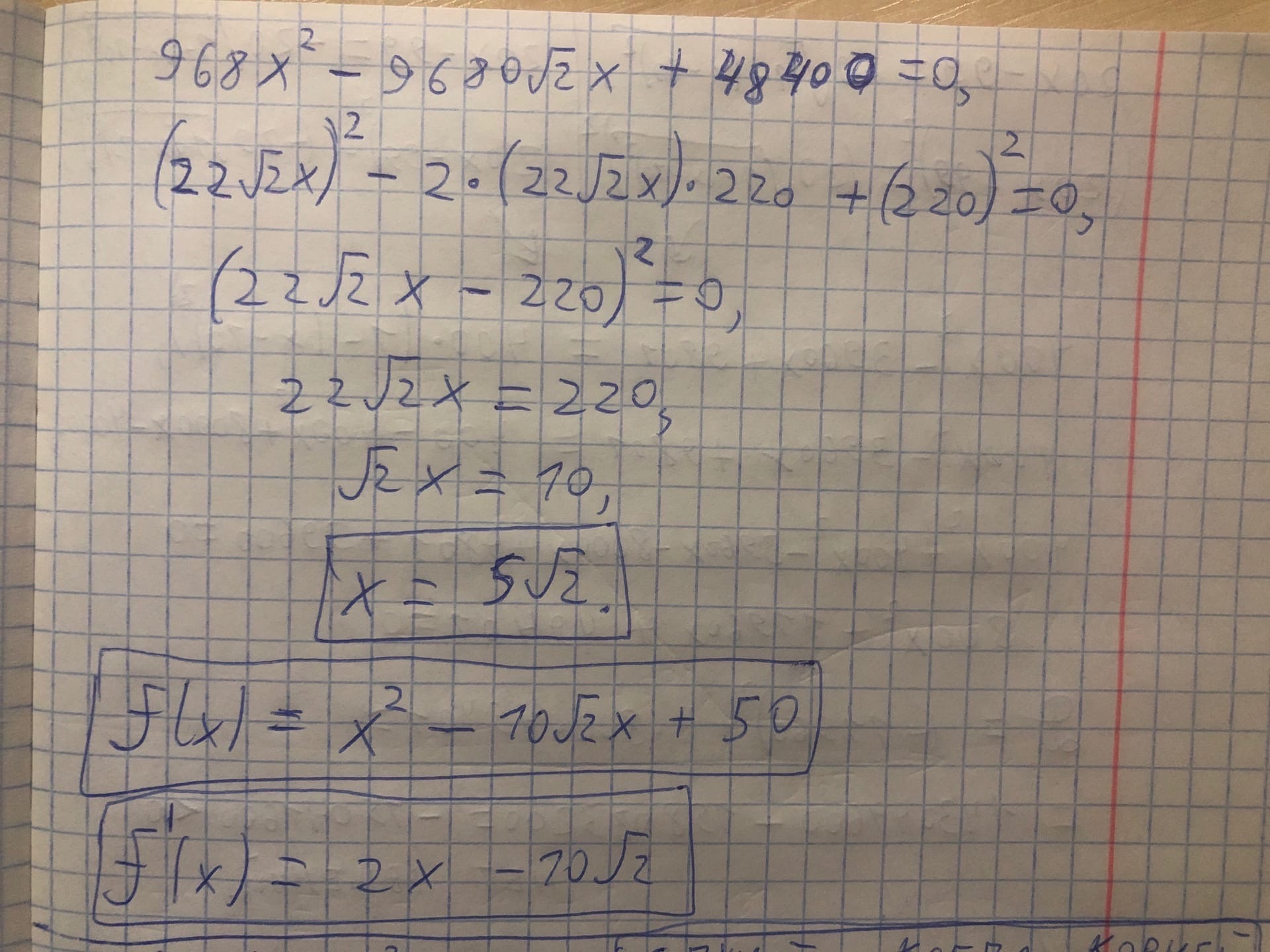
К сожалению, в GeoGebra нельзя узнать точное значение параметра c, при котором окружности касались бы, поэтому мне пришлось пойти здесь другим путём. Чтобы подсчитать параметр c, который позволяет окружностям коснуться в одной точке, я решил несложную геометрическую задачку в тетради:



Из теоремы Пифагора я нашёл оба равных катета из суммы радиусов моих окружностей, а значения этих катетов по совместительству и являются параметром c, потому что чётко видно, что большая окружность сместилась по оси Oy ровно на значение этого катета вверх, а малая окружность сместилась по оси Ox ровно на значение этого катета вправо. Так, я пришёл к выводу, что для этого случая параметр c = .

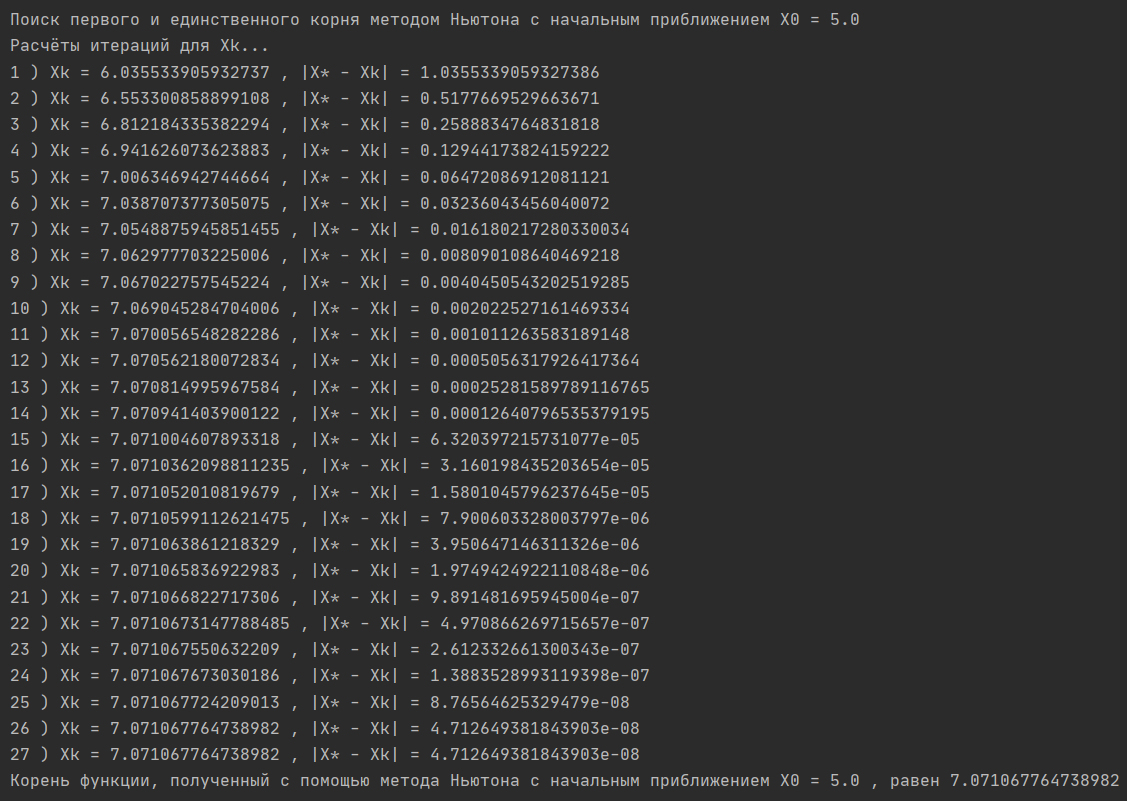
Далее я, как и в первом случае, подставил параметр c в уравнение, выразил во втором уравнении y через x и вывел уравнение функции, зависящей только от x. Я не стал считать дискриминант (хотя он, по идее, получился бы у меня равным нулю), а просто выделил полный квадрат, откуда сразу и был виден корень:





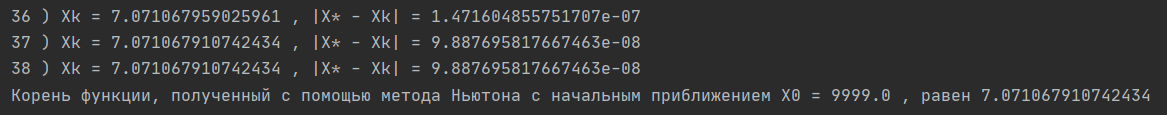
Таким образом, я теоретически нашёл тот единственный корень, к которому должен сойтись мой метод Ньютона для посчитанной относительно c = функции и её производной.

Итак, я посчитал этот единственный корень (X1 = 5), а в качестве приближения взял число X0 = 5:



Метод на данном корне сошёлся хорошо.

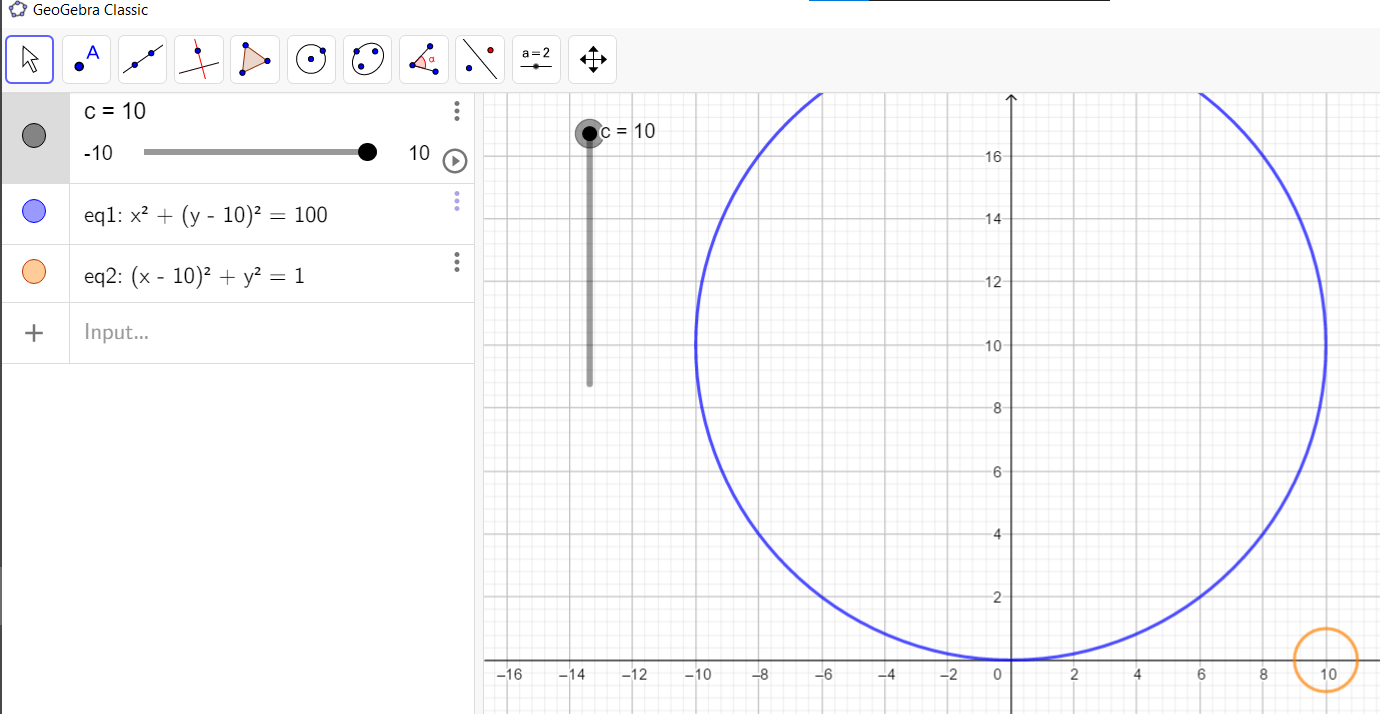
Но я также пробовал его и на более отдалённых приближениях (X0 = 9999)



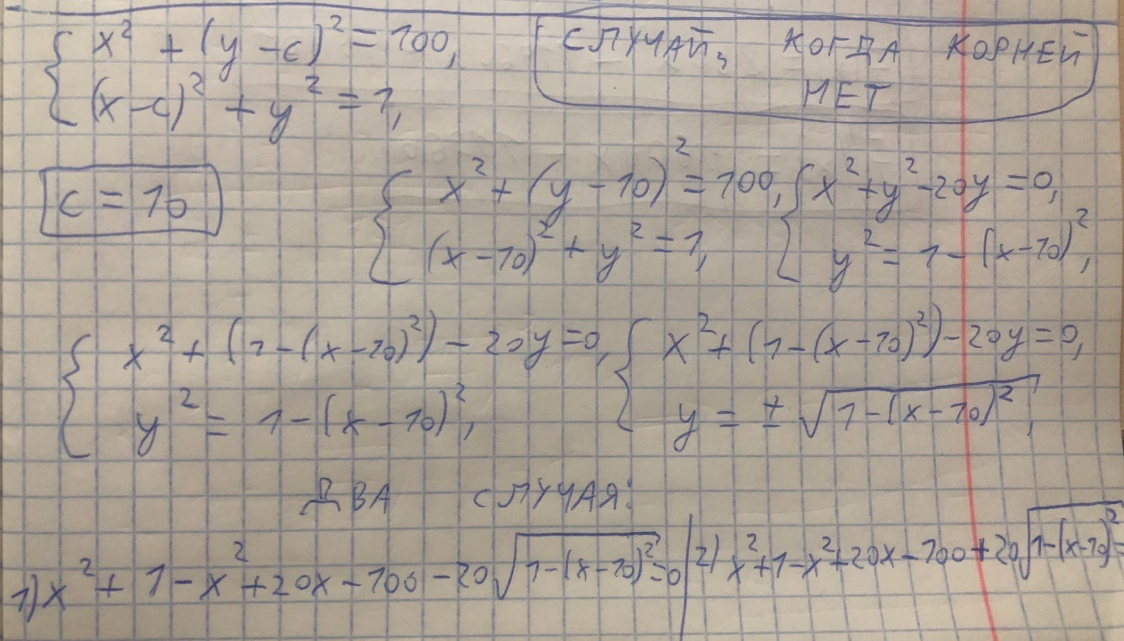
Так, для случая с одним единственным корнем Метод Ньютона также работает неплохо. Какое бы начальное приближение мы не взяли, оно сведётся к одному из двух корней (и насколько я понял, к тому из корней, который поближе). Только стоит отметить, что здесь слегка повыше погрешность вычислений (1e-8), что связано, скорее всего, с появлением корней внутри функции.

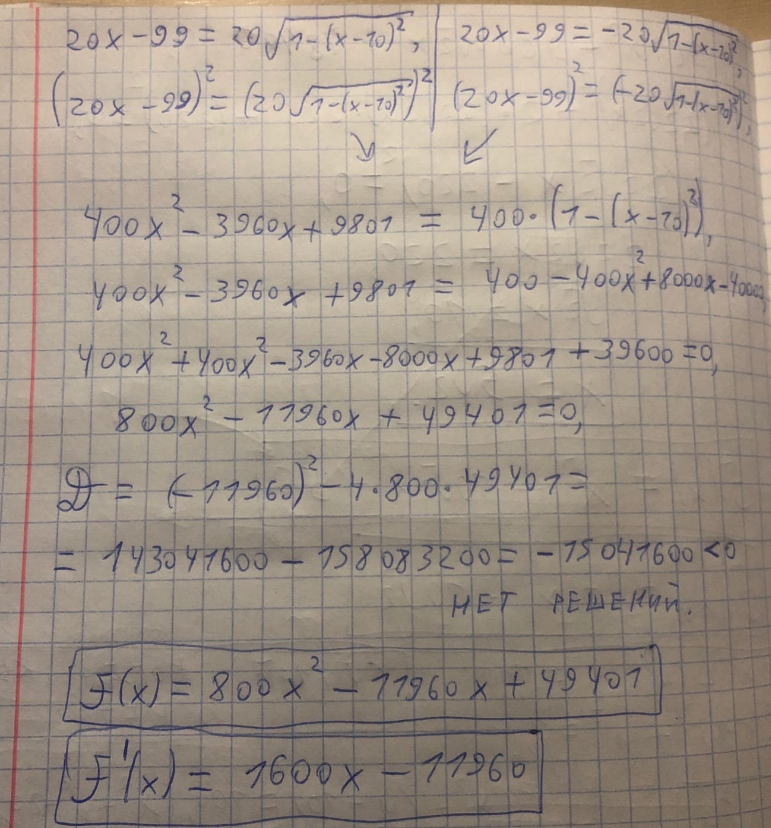
С**лучай 3 – система не имеет корней**

Итак, последний случай, который я всё же решил разобрать – это случай, в котором система не имеет корней. Для этого я выбрал какой-нибудь параметр c = 10, при котором окружности не буту пересекаться (рисунок из GeoGebra):



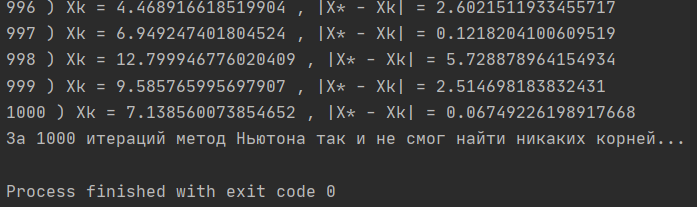
Однако, прежде чем действовать в программе, я все же посчитал это функцию и её производную (для метода Ньютона), а также убедился в том, что дискриминант отрицательный:





Так, я теоретически доказал, что корней нет. Я попробовал поискать корни методом Ньютона в программе. Изначально, я дал ему 1000 итераций (начальное приближение выбрал наугад: X0 = 5), чтобы он смог что-то найти, но ни одно значение для корня так и не сошлось:





Позже, я изменил значение 1000 на 30, чтобы графики было проще сравнить друг с другом.

Итак, подытожим все вышесказанное графиками (в письме они также будут прикреплены, под именем Task\_2\_3\_Графики\_Сходимостей.jpg)

**Графики**



Здесь NM – означает Метод Ньютона.

NM1 и NM2 показывают диаграммы сходимостей для двух корней из первого случая (когда существует только 2 корня). Видно, что графики сошлись очень быстро

На графике NM3 показана диаграмма сходимости для единственного корня из второго случая (когда существует только 1 корень). Видно, что график сошёлся, хотя не так быстро и не так точно, как предыдущие два, но опять же, вероятно, это из-за погрешностей, так как в функции второго случая были корни и большие коэффициенты.

На графике NM4 показана диаграмма сходимости для первых 30 итераций из третьего случая (корней не существует). Тут видно, что даже со временем невязка не уменьшается, а только периодически изменяется то вверх, то вниз.

В целом, это всё, что я хотел описать в своём отчёте. Я надеюсь, я достаточно описал все свои шаги, хотя отчёт и без того уже получился на 30 страниц. В целом можно сказать, что был получен полезный опыт относительно методов Бисекции, Хорд и Ньютона.