Лабораторная работа №2

Решение проблемы собственных значений. Численное решение нелинейных уравнений.

Задание 1

Написать программу, которая находит максимальное по модулю собственное значение (пару максимальных по модулю собственных значений) и соответствующий ему (им) собственный вектор (собственные векторы) матрицы A с максимально возможной точностью (в пределах обычных double чисел). Применить программу к следующим ниже входным данным. В отчете подробно изложить способ определения случая и критерия остановки итераций. Проведите экспериментальное исследование скорости работы вашей программы в зависимости от размерности матрицы, используя для тестов матрицу со случайными числами. Постройте график зависимости времени работы от размерности. Вычислите асимптотическую сложность реализованного вами алгоритма.

Будем использовать степенной метод. На каждой итерации будем проверять 3 случая:

1. $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$
2. $\lambda_1 = -\lambda_2, |\lambda_2| > |\lambda_3| \geq |\lambda_4| \geq \ldots \geq |\lambda_n|$
3. $\lambda_1, \lambda_2 -$ комплексно-сопряженная пара.

Для 1 и 2 случая критерий остановки будет следующим: $|\lambda^k - \lambda^{k+1}| < \varepsilon$, где $\lambda^i -$ собственное значение, полученное на i-ом шаге. Т.к. мы можем сравнивать только модули комплексных чисел, то для 3 случая был выбран другой критерий остановки. Если $\lambda^k = a^k + i \cdot b^k -$ собственное значение, полученное на k-ом шаге, то алгоритм остановится, если $\sqrt{((\lambda^k - \lambda^{k+1}).real)^2 + ((\lambda^k - \lambda^{k+1}).imag)^2}$, где $.real, .imag -$ действительная и мнимая части соответственно.

Решим задание для первой матрицы.

In [ ]:
```
A = [
     [1, -2, 1, 0, -1, 1, -2, 2, 0, -2],
     [0, 2, 0, 0, 2, 1, -1, -1, -1, -2],
     [0, 1, 0, -1, 1, -1, 0, -1, 1, -1],
     [-2, -1, 2, -1, 0, 0, 0, 0, 1, 0],
     [1, -2, 0, 1, 0, -2, -1, 0, 2, 2],
     [-2, -2, 0, -2, 0, 1, 1, -2, 1, 1],
     [-1, -2, -1, -1, -2, -1, -2, 1, -1, 2],
     [-2, 1, 2, -2, 0, 2, 1, -1, -2, 2],
     [0, 1, 0, 1, 1, -2, 2, 0, 1, 1],
     [0, 0, 2, -1, -1, 0, -2, 2, -1, -1]
]
y_0 = [-1,  1,  0,  2, -2, 1, 0,  0, 1, -1]
eps = 10 ** (-9)
solution = find_eigenvector(y_0, eps)
print("eigenvalue = ", solution[1])
print("eigenvector = ", solution[2])
```

```
eigenvalue =  (-0.9163675752777167+3.925408634687616j)
eigenvector =  [(-2.7438583317719933-1.6613706581663585j), (-1.0823073700518002+0.07598098201873091j), (1.0521868
703941066+2.1259307071669014j), (-2.144738857419304+2.7103568874430017j), (1.6028188029245047-0.5033906229812312j
), (-0.3274532009325066+3.925408634687616j), (1.8094259221586007-1.221477336720101j), (-1.0341300815818908+1.3583
351585553747j), (1.7854908478408666+1.1470283997755957j), (-1.5686324056813947-1.1960845281879557j)]
```

Теперь найдем найдем собственное значение и собственный вектор для второй матрицы.

In [ ]:
```
A = [
     [-1, 1, -1, 0, -1, 0, -1, 1, 1, -1, 0, -1, -1, 1, 0, 0, 1, 1, 1, 1],
     [-1, 0, -1, 1, -1, 0, 0, 0, 0, -1, 0, 0, -1, 1, 0, -1, 1, -1, -1, 0],
     [1, 0, -1, 1, 0, 1, -1, -1, -1, 0, -1, -1, 1, -1, 1, 1, -1, 1, -1, 0],
     [-1, 1, 0, 0, -1, 0, 0, -1, 0, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 0],
     [1, 0, -1, 0, 0, -1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, -1, 0, 0, 1],
     [0, 0, 0, 0, -1, 1, 1, 0, 0, 1, 1, 0, -1, 0, 1, 1, 0, 1, 0, 0],
     [-1, 0, 1, 1, 1, -1, -1, 0, -1, 1, -1, -1, -1, 0, -1, 0, 0, 0, -1, 1],
     [0, 0, -1, -1, 0, 1, 1, 1, 1, -1, 0, 0, -1, 1, 1, 1, 1, 0, 0, -1],
     [0, 0, 1, 1, 0, 1, 1, 0, 1, -1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1],
     [0, -1, 0, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, -1, 0, 0, 1, 1],
     [1, -1, 1, -1, -1, -1, 1, 0, -1, 0, 1, 1, -1, 0, 1, 1, 1, 0, 0, 0],
     [0, 1, 0, 0, -1, 0, 1, 0, 1, 0, 0, 1, 1, -1, -1, 0, -1, 1, 1, -1],
     [-1, -1, -1, -1, 0, 1, -1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, 0, -1, 0],
     [-1, 0, 1, 0, -1, 0, 0, 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 0],
     [1, -1, 0, -1, -1, 0, -1, -1, 0, 0, 1, 0, 1, 1, -1, 1, 0, 0, -1, 0],
```

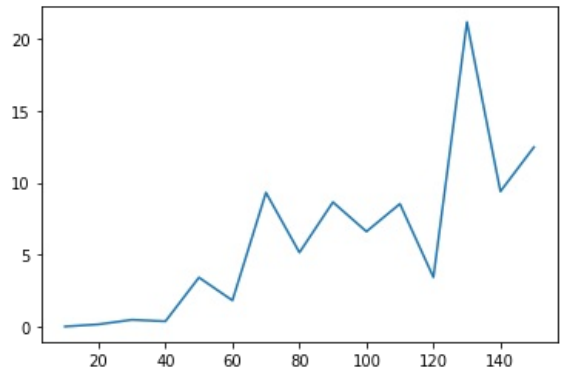
```
     [-1, -1, 1, 0, -1, 1, 1, -1, 1, 0, 0, -1, 1, -1, -1, 0, 0, 1, 1, 1],
     [0, 0, -1, 0, 0, 0, 0, -1, 1, 1, 0, -1, 1, -1, 0, 0, 0, -1, -1, 1],
     [-1, 0, -1, -1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, 0, -1, 0, -1],
     [-1, 0, 1, 0, 0, 0, 0, -1, 1, -1, 1, -1, 0, -1, -1, 1, 0, 1, 0, 0],
     [0, -1, -1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 0, -1, -1, 0, 1, 0, -1, -1]
     ];
y_0 = [-1,  1,  0,  2, -2, 1, 0,  0, 1, -1, -1,  1,  0,  2, -2, 1, 0,  0, 1, -1]
eps = 10 ** (-9)
solution = find_eigenvector(y_0, eps)
print("eigenvalue = ", solution[1])
print("eigenvector = ", solution[2])
```

```
eigenvalue =  (3.852317078696089+7.390069824471696e-19j)
eigenvector =  [(-1.8640644583456378e-13-1.2758639903638388e-19j), (5.297984273511247e-13+3.5806495230694324e-19j
), (-2.6501023597802487e-13-1.7820770191277606e-19j), (-6.412648190234904e-13-4.35925372668421e-19j), (4.17887946
4689089e-13+2.8541720238718894e-19j), (-5.990763440877345e-13-4.0478131583161674e-19j), (4.5230486023228877e-13+3
.034446704110329e-19j), (3.4468955467659157e-15+1.2032220963588114e-21j), (-8.810729923425242e-13-5.9645720906160
62e-19j), (2.9726221484338566e-14+1.8065443793815073e-20j), (-2.4358293160275934e-13-1.663102435047839e-19j), (-6
.417089082333405e-13-4.3190105688813284e-19j), (3.552713678800501e-13+2.378145679415153e-19j), (1.087130385712953
3e-12+7.390069824471696e-19j), (2.155220446553585e-14+1.298825184709872e-20j), (-9.086065233532281e-13-6.1447591
31294739e-19j), (4.257011410047085e-15+5.501425872018146e-21j), (-2.7844393457598926e-13-1.8504509937213821e-19j)
, (-7.451816941284051e-13-5.06778796075127e-19j), (-2.964295475749168e-13-1.9995201553441706e-19j)]
```

Теперь проанализируем зависимость времени работы алгоритма от размера матрицы.

In [ ]:

```
eps = 10 ** (-3)
times = []
vals = []
for size in range(2, 151):
  A = [[0] * size for i in range(size)]
  y_0 = []
  if (size % 10 == 0):
    for i in range(size):
      y_0.append(random.randint(-5, 5))
      while (y_0[i] == 0):
        y_0[i] = random.randint(-5, 5)
      for j in range(size):
        A[i][j] = random.randint(-5, 5)
    #print(size)
    start_time = time.time()
    solution = find_eigenvector(y_0, eps)
    #print("eigenvalue = ", solution[1])
    #print("eigenvector = ", solution[2])
    #print("time = ", time.time() - start_time)
    times.append(time.time() - start_time)
    vals.append(size)
n = len(vals)
plt.plot(vals, times)
plt.show()
```



Из графика видим, что с размером матрицы время работы увеличивается. Но так же есть и "скачки", где время работы меньше, чем у матрицы с меньшим размером. Так происходит потому, что сходимость алгоритма зависит от того, насколько максимальное собственное значение отличается от остальных.

Задание 2

Написать программу, реализующую QR-алгоритм нахождения всех собственных значений матрицы A с максимально возможной точностью. Применить программу к матрицам из первого задания. Проведите экспериментальное исследование скорости работы вашей программы в зависимости от размерности матрицы, используя для тестов матрицу со случайными числами. Постройте график зависимости времени работы от размерности. Вычислите асимптотическую сложность реализованного вами алгоритма.

Будем выполнять итерации QR алгоритма, пока под главной диагональю не будет достаточное количество нулей.

После этого будем искать собственные значения. В полученной матрице будем идти по главной диагонали и рассматривать 2 случая:

1. На i-ом шаге под главной диагональю стоит 0. Значит мы нашли действительное собственное значение.
2. На i-ом шаге под главной диагональю стоит число, не равное 0. Значит для этого блока $2 * 2$ будем искать собственные значения по определению.

Решим для 1ой матрицы.

In [ ]:
```
A = [
     [1, -2, 1, 0, -1, 1, -2, 2, 0, -2],
     [0, 2, 0, 0, 2, 1, -1, -1, -1, -2],
     [0, 1, 0, -1, 1, -1, 0, -1, 1, -1],
     [-2, -1, 2, -1, 0, 0, 0, 0, 1, 0],
     [1, -2, 0, 1, 0, -2, -1, 0, 2, 2],
     [-2, -2, 0, -2, 0, 1, 1, -2, 1, 1],
     [-1, -2, -1, -1, -2, -1, -2, 1, -1, 2],
     [-2, 1, 2, -2, 0, 2, 1, -1, -2, 2],
     [0, 1, 0, 1, 1, -2, 2, 0, 1, 1],
     [0, 0, 2, -1, -1, 0, -2, 2, -1, -1]
]

n = len(A)
while (Zeros() < (n - 1) // 2):
  MakingForm()
```

In [ ]:
```
res = SolutionQR()
eps = 10 ** (-9)
while (1):
  MakingForm()
  new_res = SolutionQR()
  n = len(res)
  mx = 0
  for i in range(n):
    diff = res[i] - new_res[i]
    val = (diff.real ** 2 + diff.imag ** 2) ** 0.5
    mx = max(mx, val)
    res[i] = new_res[i]

  if (mx < eps):
    break
print("eigenvalue:")
for i in range(len(res)):
  print(res[i])
```

```
eigenvalue:
(-0.9163675754652342-3.925408634874052j)
(-0.9163675754652337+3.925408634874052j)
(1.3620793201808297-3.66685855409186j)
(1.36207932018083+3.66685855409186j)
(-3.4157126078424045+0j)
(3.1131615107820485+0j)
(-0.7547394430730903-1.2607574159110637j)
(-0.75473944307309+1.2607574159110637j)
(0.9972017001139122+0j)
(-0.07659520633857221+0j)
```

Аналогично для 2ой.

In [ ]:
```
A = [
     [-1, 1, -1, 0, -1, 0, -1, 1, 1, -1, 0, -1, -1, 1, 0, 0, 1, 1, 1, 1],
     [-1, 0, -1, 1, -1, 0, 0, 0, 0, -1, 0, 0, -1, 1, 0, -1, 1, -1, -1, 0],
     [1, 0, -1, 1, 0, 1, -1, -1, -1, 0, -1, -1, 1, -1, 1, 1, -1, 1, -1, 0],
     [-1, 1, 0, 0, -1, 0, 0, -1, 0, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 0],
     [1, 0, -1, 0, 0, -1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, -1, 0, 0, 1],
     [0, 0, 0, 0, -1, 1, 1, 0, 0, 1, 1, 0, -1, 0, 1, 1, 0, 1, 0, 0],
     [-1, 0, 1, 1, 1, -1, -1, 0, -1, 1, -1, -1, -1, 0, -1, 0, 0, 0, -1, 1],
     [0, 0, -1, -1, 0, 1, 1, 1, 1, -1, 0, 0, -1, 1, 1, 1, 1, 0, 0, -1],
     [0, 0, 1, 1, 0, 1, 1, 0, 1, -1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1],
     [0, -1, 0, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, -1, 0, 0, 1, 1],
     [1, -1, 1, -1, -1, -1, 1, 0, -1, 0, 1, 1, -1, 0, 1, 1, 1, 0, 0, 0],
     [0, 1, 0, 0, -1, 0, 1, 0, 1, 0, 0, 1, 1, -1, -1, 0, -1, 1, 1, -1],
     [-1, -1, -1, -1, 0, 1, -1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, 0, -1, 0],
     [-1, 0, 1, 0, -1, 0, 0, 1, -1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 0],
     [1, -1, 0, -1, -1, 0, -1, -1, 0, 0, 1, 0, 1, 1, -1, 1, 0, 0, -1, 0],
     [-1, -1, 1, 0, -1, 1, 1, -1, 1, 0, 0, -1, 1, -1, -1, 0, 0, 1, 1, 1],
     [0, 0, -1, 0, 0, 0, 0, -1, 1, 1, 0, -1, 1, -1, 0, 0, 0, -1, -1, 1],
```

```
    [-1, 0, -1, -1, -1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, 0, -1, 0, -1],
    [-1, 0, 1, 0, 0, 0, 0, -1, 1, -1, 1, -1, 0, -1, -1, 1, 0, 1, 0, 0],
    [0, -1, -1, 1, -1, 1, -1, -1, -1, 1, 1, -1, 0, -1, -1, 0, 1, 0, -1, -1]
    ];


n = len(A)
while (Zeros() < (n - 1) // 2):
  MakingForm()
```

In [ ]:

```
res = SolutionQR()
eps = 10 ** (-7)
while (1):
  MakingForm()
  new_res = SolutionQR()
  n = len(res)
  mx = 0
  for i in range(n):
    diff = res[i] - new_res[i]
    val = (diff.real ** 2 + diff.imag ** 2) ** 0.5
    mx = max(mx, val)
    res[i] = new_res[i]

  if (mx < eps):
    break
print("eigenvalues:")
for i in range(len(res)):
  print(res[i])
```

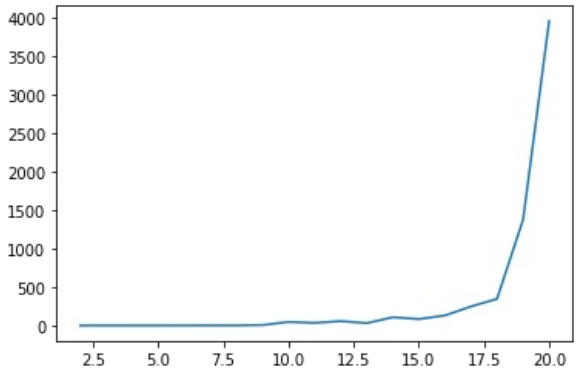
```
eigenvalues:
(3.8523170786950205+0j)
(-3.6692058718213003+0j)
(-3.232785229295595+0j)
(1.3329305475464388-2.8056368468001898j)
(1.3329305475464392+2.8056368468001898j)
(-1.7228524817928805-2.5277630088675447j)
(-1.72285248179288+2.5277630088675447j)
(2.9919297758421846+0j)
(-0.769186903976937-2.7312339648501607j)
(-0.7691869039769366+2.7312339648501607j)
(0.09247019805045556-2.8253194590603123j)
(0.09247019805045589+2.8253194590603123j)
(-2.2964069288872704+0j)
(2.3057352521444674+0j)
(1.6422964860573723+0j)
(0.8568128341643745-1.0670890280678822j)
(0.8568128341643747+1.0670890280678822j)
(-0.813381199229594+0j)
(-0.18042387574402155-0.3117167722500945j)
(-0.18042387574402155+0.3117167722500945j)
```

Теперь проанализируем зависимость времени работы алгоритма от размера матрицы.

In [ ]:

```
eps = 10 ** (-3)
times = []
vals = []
for size in range(2, 21):
  A = [[0] * size for i in range(size)]
  for i in range(size):
    for j in range(size):
      A[i][j] = random.randint(-5, 5)
  #print("size = ", size)
  start_time = time.time()
  n = len(A)
  while (Zeros() < (n - 1) // 2):
    MakingForm()
  res = SolutionQR()
  while (1):
    MakingForm()
    new_res = SolutionQR()
    n = len(res)
    mx = 0
    for i in range(n):
      diff = res[i] - new_res[i]
      val = (diff.real ** 2 + diff.imag ** 2) ** 0.5
      mx = max(mx, val)
      res[i] = new_res[i]
    if (mx < eps):
      break
  times.append(time.time() - start_time)
  vals.append(size)
n = len(vals)
plt.plot(vals, times)
```

```
plt.show()
```



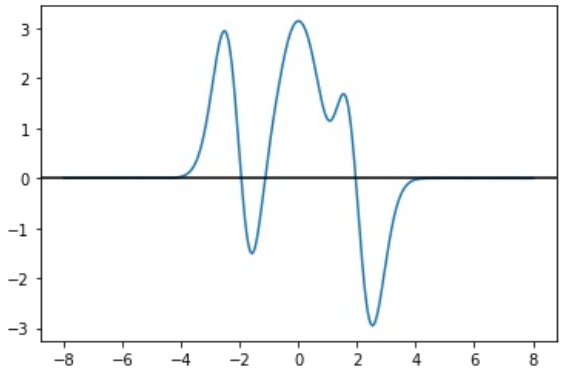
Из графика видим, что с размером матрицы время работы увеличивается. Но так же есть и "скачки", где время работы меньше, чем у матрицы с меньшим размером. Так происходит потому, что сходимость алгоритма зависит от того, насколько сильно различаются собственные значения.

Задание 3

Дано нелинейное уравнение $\frac{(x^9+\pi)cos(ln(x^2+1))}{e^{x^2}} - \frac{x}{2022} = 0.$

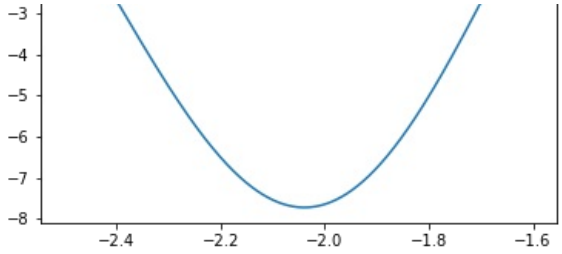
Отделить все его корни. Обосновать (не обязательно доказать строго) единственность каждого корня на отрезке, отсутствие других корней. Методом бисекции сузить отрезки отделенности корней до размера не более 10^{-4}. Методом Ньютона найти все корни с максимально возможной точностью. В отчет включить количество итераций обоих методов.

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
  return (x ** 9 + np.pi) * (np.cos(np.log(x**2 + 1))) / np.exp(x**2) - x/2022
y = lambda x: f(x)


fig = plt.subplots()
x = np.linspace(-8, 8,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
plt.show()
```



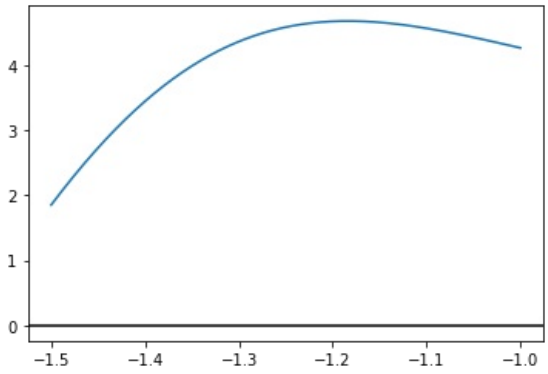
Из графика видно, что корни находятся на промежутках $[-2.5, -1.6], [-1.5, -1], [1.6, 2.5]$.

In [20]:
```
y = lambda x: fp(x)
fig = plt.subplots()
x = np.linspace(-2.5, -1.6,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
plt.show()
```



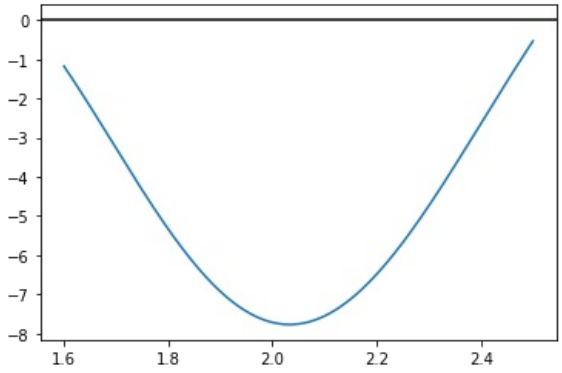
In [ ]:

```
y = lambda x: fp(x)
fig = plt.subplots()
x = np.linspace(-1.5, -1,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
plt.show()
```



In [ ]:

```
y = lambda x: fp(x)
fig = plt.subplots()
x = np.linspace(1.6, 2.5,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
plt.show()
```



Как видим, производная на этих отрезках не меняет свой знак, значит на них по 1 корню.

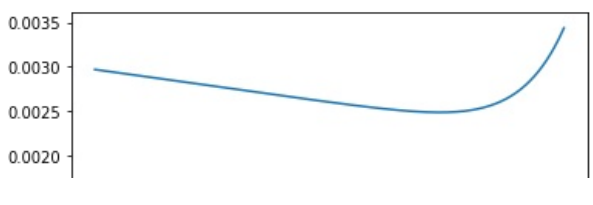
Так же возможно есть корни на промежутках $[-6, -4.5], [4, 5]$. Чтобы проверить, есть ли они на них, посмотрим как выглядит график функции на этих промежутках.

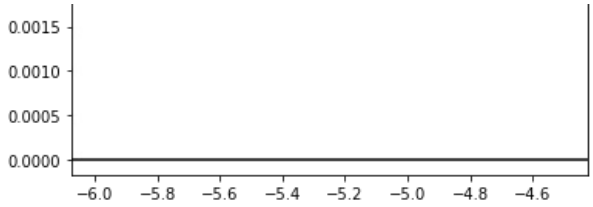
In [ ]:

```
x = np.linspace(-6, -4.5,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
```

Out[ ]:

```
<matplotlib.lines.Line2D at 0x7f0dfe43c490>
```



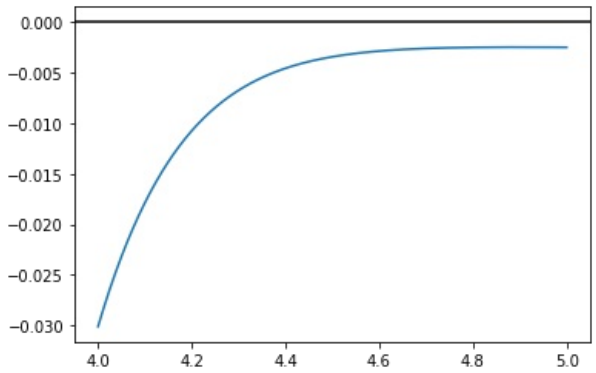


In [ ]:

```
x = np.linspace(4, 5,100000)
plt.plot(x, y(x))
ax = plt.gca()
ax.axhline(y=0, color='k')
```

Out[ ]:

```
<matplotlib.lines.Line2D at 0x7f0dfe2ea5d0>
```



Из графиков видно, что на промежутках $[-6, -4.5], [4, 5]$ корней нет.

In [21]:

```
a = -2.5
b = -1.6
print("old borders: [", a, ", ", b, "]", sep = '')
borders = bin_search(-3, -1.5, 10**-4)
a = borders[0]
b = borders[1]
print("iter for search new borders: ", borders[2])
print("new borders: [", a, ", ", b, "]", sep = '')
x, iter = Newton(a, b, 10**-30)
print("solution: ")
print("iter = ", iter)
print("x =", x)
print("f(x) =", f(x))
```

```
old borders: [-2.5, -1.6]
iter for search new borders:  14
new borders: [-1.951995849609375, -1.951904296875]
solution:
iter =  3
x = -1.9519128361044704
f(x) = 4.979740578225922e-16
```

In [22]:

```
a = -1.5
b = -1
print("old borders: [", a, ", ", b, "]", sep = '')
borders = bin_search(a, b, 10**-4)
a = borders[0]
b = borders[1]
print("iter for search new borders: ", borders[2])
print("new borders: [", a, ", ", b, "]", sep = '')
x, iter = Newton(a, b, 10**-30)
print("solution: ")
print("iter = ", iter)
print("x =", x)
print("f(x) =", f(x))
```

```
old borders: [-1.5, -1]
iter for search new borders:  13
new borders: [-1.13580322265625, -1.1357421875]
solution:
iter =  3
x = -1.1357564642357232
f(x) = -3.2526065174565133e-18
```

In [23]:
```
a = 1.6
b = 2.5
print("old borders: [", a, ", ", b, "]", sep = '')
borders = bin_search(a, b, 10**-4)
a = borders[0]
b = borders[1]
print("iter for search new borders: ", borders[2])
print("new borders: [", a, ", ", b, "]", sep = '')
x, iter = Newton(a, b, 10**-30)
print("solution: ")
print("iter = ", iter)
print("x =", x)
print("f(x) =", f(x))
```

```
old borders: [1.6, 2.5]
iter for search new borders:  14
new borders: [1.9518920898437502, 1.9519470214843753]
solution:
iter =  3
x = 1.951914830924147
f(x) = -6.766505758482033e-16
```

Как видим, метод Ньютона нашел ответ за 3 итерации, что является хорошим результатом.

Далее представлены функции, которые использовались в коде выше.

Метод бисекции для уточнения границ промежутков.

In [17]:
```
def bin_search(a, b, eps):
  iter = 0
  while(b - a > eps):
    m = (b + a) / 2
    iter += 1
    if (f(a) * f(m) > 0):
      a = m
    else:
      b = m
  return [a, b, iter]
```

Поиск значения функции в точке x.

In [16]:
```
def fp(x):
  fraction1 = -2 * np.exp(-(x**2))*(x**9 + np.pi) * x * np.sin(np.log(x**2 + 1)) / (x**2 + 1)
  fraction2 = -2 * np.exp(-(x**2))*(x**9 + np.pi) * x * np.cos(np.log(x**2 + 1))
  fraction3 = 9 * np.exp(-(x**2)) * (x ** 8) * np.cos(np.log(x**2 + 1)) - 1 /2022
  return fraction1 + fraction2 + fraction3
```

Метод Ньютона для поиска решений.

In [15]:
```
def Newton(a, b, eps):
  x0 = b
  x1 = (a + b) /2
  iter = 0
  while(abs(x1-x0) > eps):
    x0 = x1
    x1 = x0 - f(x0)/fp(x0)
    iter += 1
  return x1, iter
```

In [1]:
```
import random
import time
import matplotlib.pyplot as plt
```

Перемножение матриц.

In [14]:
```
def MultiplyMatrix(A, B):
  n = len(A)
  C = [[0] * n for i in range(n)]
  for i in range(n):
    for j in range(n):
      for l in range(n):
        C[i][j] += A[i][l] * B[l][j]
```

```
  return C
```

Нормирование вектора и получение нормы вектора.

In [2]:
```
def normalize(a):
  mx = get_norm(a)
  res = []
  n = len(a)
  for i in range(n):
    x = a[i]
    res.append(x / mx)
  return res


def get_norm(a):
  mx = 0
  n = len(a)
  for i in range(n):
    x = a[i]
    mx = max(mx, abs(x))
  return mx
```

Умножение матрицы на вектор.

In [4]:
```
def Multiply(A, b):
  n = len(b)
  res = [0] * n
  for i in range(n):
    for j in range(len(A[i])):
      res[i] += A[i][j] * b[j]
  return res
```

Поиск собственных значений степенным методом.

In [5]:
```
import math
def find_eigenvector(y, eps):
  n = len(y)
  u1 = []
  yi = []
  yi_1 = []
  for i in range(n):
    u1.append(y[i])
    yi.append(y[i])
    yi_1.append((y[i]))
  l1 = 2**31
  l2 = l1
  l3 = complex(l1)
  iter = 0
  case1 = 1
  case2 = 1
  case3 = 1
  diff1 = l1
  diff2 = l1
  diff3 = l1
  case_solution = -1
  eigenvalue = 0
  eigenvector = []
  while(case1 or case2 or case3):
    iter+=1
    if case1:
      res = solve_case1(A, u1, l1)
      if (diff1 < abs(l1 - res[1]) and iter > 20):
        case1 = 0
      diff1 = abs(l1 - res[1])
      l1 = res[1]
      if (res[0] == 1):
        case_solution = 1
        eigenvalue = l1
        eigenvector = u1
        break

    if case2:
      res = solve_case2(A, yi, l2)
      if (diff2 < abs(l2 - res[1]) and iter > 20):
        case2 = 0
      diff2 = abs(l2 - res[1])
      l2 = res[1]
      if (res[0] == 1):
        case_solution = 2
        eigenvalue = l2
        eigenvector = res[3]
        break
```

```
      else:
        for i in range(n):
          yi[i] = res[2][i]

    if case3:
      res = solve_case3(A, yi_1, l3)
      if (iter > 700):
        case3 = 0
      if (((l3.real - res[1].real) ** 2 + abs(l3.imag - res[1].imag) ** 2) ** 0.5 < eps):
        case_solution = 3
        eigenvalue = res[1]
        eigenvector = res[2]
        break
      l3 = res[1]
      for i in range(n):
        yi_1[i] = res[3][i]
  return [case_solution, eigenvalue, eigenvector]
```

Степенной метод, 1 случай

In [6]:

```
def solve_case1(A, u, l):
  v = Multiply(A, u)
  j = 0
  n = len(v)
  for i in range(n):
    if math.fabs(u[j]) < math.fabs(u[i]) and v[i] != 0:
      j = i
  l1 = v[j] / u[j]
  for i in range(n):
    u[i] = v[i] / v[j]
  if abs(l1 - l) < eps:
    return [1, l1]
  return [0, l1]
```

Степенной метод, 2 случай

In [7]:

```
def solve_case2(A, yi, l):
  yi1 = Multiply(A, yi)
  yi2 = Multiply(A, yi1)
  j = 0
  n = len(yi2)
  norm_yi2 = 0
  for i in range(n):
    norm_yi2 = max(norm_yi2, yi2[i])
    if math.fabs(yi[j]) < math.fabs(yi[i]):
      j = i
  l1 = (abs(yi2[j] / yi[j]))**0.5
  v = [0] * n
  norm = 0
  for i in range(n):
    v[i] = yi2[i] + l1 * yi[i] # собственный вектор
    norm += v[i] ** 2
  norm = norm ** 0.5
  for i in range(n):
    v[i] /= norm
  for i in range(n):
    yi2[i] /= norm_yi2

  if abs(l1 - l) < eps:
    return [1, l1, yi2, v]
  return [0, l1, yi2, v]
```

Степенной метод, 3 случай

In [24]:

```
def solve_case3(A, ui_1, l_last):
  n = len(ui_1)

  vi = Multiply(A, ui_1)
  ui = normalize(vi)

  vi1 = Multiply(A, ui)
  ui1 = normalize(vi1)

  vi2 = Multiply(A, ui1)
  ui2 = normalize(vi2)

  j = -1
  mx = 0
```

```
  for i in range(n):
    if (vi1[i] != 0 and
        abs(vi[i] * vi2[i] * get_norm(vi1) - vi1[i] * vi1[i] * get_norm(vi)) != 0
        and abs(ui_1[i] * vi1[i] - (ui[i] ** 2)*get_norm(vi)) > mx
        and abs(ui_1[i] * vi1[i] - (ui[i] ** 2)*get_norm(vi)) * vi1[i] != 0):
      mx = abs(ui_1[i] * vi1[i] - (ui[i] ** 2)*get_norm(vi))
      j = i

  r = abs(vi[j] * vi2[j] * get_norm(vi1) - vi1[j] * vi1[j] * get_norm(vi))
  r /= mx
  r = r ** 0.5

  cosf = vi2[j] * get_norm(vi1) + r * r * ui[j]
  cosf /= 2 * r * vi1[j]

  sinf = (1 - cosf ** 2) ** 0.5
  lyambda = complex(r * cosf, r * sinf)
  v = [0] * n
  for i in range(n):
    v[i] = vi1[i] - lyambda * ui[i]
  nev = Multiply(A, v)
  norm = 0
  for i in range(n):
    nev[i] -= lyambda * v[i]
    norm += nev[i] ** 2
  return [0, lyambda, v, ui2, norm ** 0.5]
```

Получение транспонированной матрицы.

In [9]:

```
def T(Q, n):
  QT = [[0] * n for i in range(n)]
  for i in range(n):
    for j in range(n):
      QT[i][j] = Q[j][i]
  return QT
```

Получение единичной матрицы.

In [10]:

```
def IdentityMatrix(n):
  E = [[0] * n for i in range(n)]
  for i in range(n):
    E[i][i] = 1
  return E
```

Приведение к нужной форме, для QR алгоритма.

In [11]:

```
def MakingForm():
  global A
  n = len(A)
  Q = IdentityMatrix(n)

  for j in range(n - 1):
    for i in range(j + 1, n):
      # зануляем [i][j] элемент
      if (A[i][j] == 0):
        continue
      det = (A[i][j] **2 + A[j][j] ** 2) ** 0.5
      sinf = A[i][j] / det
      cosf = A[j][j] / det

      Givens = IdentityMatrix(n)
      Givens[i][i] = cosf
      Givens[j][j] = cosf
      Givens[i][j] = -sinf
      Givens[j][i] = sinf
      A = MultiplyMatrix(Givens, A)
      Q = MultiplyMatrix(Q, T(Givens, n))
  A = MultiplyMatrix(A, Q)
```

Поиск количества нулей под главной диагональю.

In [12]:

```
def Zeros():
  global A
  n = len(A)
  eps = 10 ** (-15)
  zeros = 0
  for i in range(n - 1):
    if (abs(A[i+1][i]) < eps):
      zeros += 1
  return zeros
```

Поиск решения на приведенной к нужной форме матрице.

In [13]:

```
def SolutionQR():
  global A
  n = len(A)
  eps = 10 ** (-9)
  result = []
  i = 0
  while i < n:
    if (i == n - 1 or (i + 1 < n and abs(A[i + 1][i]) < eps)):
      result.append(complex(A[i][i]))
      i += 1
    else:
      D = (A[i][i] + A[i + 1][i + 1]) ** 2 - 4 * (A[i][i] * A[i + 1][i + 1] - A[i + 1][i] * A[i][i + 1])
      l1 = (A[i][i] + A[i + 1][i + 1] - complex(D) ** 0.5) / 2
      l2 = (A[i][i] + A[i + 1][i + 1] + complex(D) ** 0.5) / 2
      i += 2
      result.append(l1)
      result.append(l2)
  return result
```