Правительство Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования
"Национальный исследовательский университет
"Высшая школа экономики"

Московский институт электроники и математики Национального исследовательского университета "Высшая школа экономики"

Департамент прикладной математики

# ОТЧЕТ
По лабораторной работе №3
на тему
«РЕШЕНИЕ СИСТЕМ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ ИТЕРАЦИОННЫМИ МЕТОДАМИ»

| ФИО студента | Номер группы | Дата | Вариант |
|---|---|---|---|
| Ткаченко Никита Андреевич | БПМ211 | 21.03.2024 | 4.1.21, 4.2.4, 5.1.21, 5.2 |

Москва – 2024 г.

# 4.1. Решение систем нелинейных уравнений

**Задача 4.1.** Найти с точностью $\varepsilon = 10^{-6}$ все корни системы нелинейных уравнений

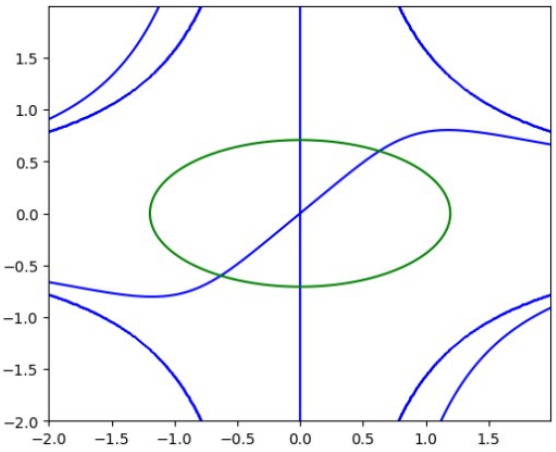
$$f_1(x_1, x_2) = 0,$$
$$f_2(x_1, x_2) = 0,$$

используя метод Ньютона для системы нелинейных уравнений.

4.1.21

$$\begin{cases} \tan(x_1 x_2) - x_1^2 = 0 \\ 0.7x_1^2 + 2x_2^2 - 1 = 0 \end{cases}$$

Локализуем графически корни данной системы уравнений:




Как видно, корни (пересечения графиков) находятся в пределах значений:

(0, 0.5); (0, -0.5); (0.5, 0.5); (-0.5, -0.5)

Реализуем метод Ньютона для решения системы нелинейных уравнений (алгебраически):

Нам дана система нелинейных уравнений вида *f1(x1, .., xm) = 0, f2(x1, .., xm) = 0*
Заменим в ней *fi(x)* на линейную часть ее разложения в ряд Тейлора, т.е.

```
fi(x) = fi(xn) + Sum[dfi(xn)/dxj * (xj – xnj), j=1, m] ,
      где xn это приближенное решение
```

Выразив отсюда fi(xn) (т.е. искомое приближенное решение), записав производную в матричной форме, и выразив из получившегося xn, получим:

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \left(\boldsymbol{f}'\left(\boldsymbol{x}^{(n)}\right)\right)^{-1} \boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right).$$

Из-за трудоемкости нахождения обратной матрицы можно упростить формулу, вместо решив эквивалентную СЛАУ:

$$\boldsymbol{f}'(\boldsymbol{x}^{(n)})\Delta\boldsymbol{x}^{(n+1)} = -\boldsymbol{f}(\boldsymbol{x}^{(n)}) \qquad (7.16)$$

относительно поправки $\Delta\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n+1)} - \boldsymbol{x}^{(n)}$. Затем полагают

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \Delta\boldsymbol{x}^{(n+1)}. \qquad (7.17)$$

, однако для нашего случая можно использовать и первую формулу, что я и делаю:

```python
def newton_method(F, eps, initial):
    x_initial = np.array(initial)
    iteration_number = 0
    jacobian = F.jacobian(X)
    while np.linalg.norm(np.array(F.subs(zip(X, x_initial)),dtype=float).flatten()) > eps:
        iteration = np.array(
            (jacobian.inv()*F) #Final equation from the textbook
            .subs(zip(X, x_initial)), dtype=float).flatten()
        x_initial = x_initial - iteration # Iteration
        iteration_number += 1
    return x_initial, iteration_number
```

Найдем все корни уравнения выше, используя написанный метод Ньютона:

```python
print(f"Newton's method: {newton_method(F1, 0.000001, [0, -0.5])}; Scipy's fsolve: {solve_F1_scipy([0, -0.5])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [0, 0.5])}; Scipy's fsolve: {solve_F1_scipy([0, 0.5])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [0.6, 0.6])}; Scipy's fsolve: {solve_F1_scipy([0.6, 0.6])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [-0.6, -0.6])}; Scipy's fsolve: {solve_F1_scipy([-0.6, -0.6])}")
```

Executed at 2024.03.22 16:54:29 in 10s 739ms

```
Newton's method: (array([ 0.        , -0.70710678]), 4); Scipy's fsolve: [-3.32755171e-26 -7.07106781e-01]
Newton's method: (array([0.        , 0.70710678]), 4); Scipy's fsolve: [1.58173704e-26 7.07106781e-01]
Newton's method: (array([0.63102535, 0.60052681]), 3); Scipy's fsolve: [0.63102535 0.60052681]
Newton's method: (array([-0.63102535, -0.60052681]), 3); Scipy's fsolve: [-0.63102535 -0.60052681]
```

Видим, что точность результатов близка к точности, полученной после решения встроенными фунциями *(scipy.optimize.fsolve)*

# 4.2.

| | | | |
|---|---|---|---|
| 4.2.4 | $x_1 - x_2^3 + 0.5\alpha$ | $\cos(2x_1) - x_2 - \alpha$ | 0, 1, -0.5 |

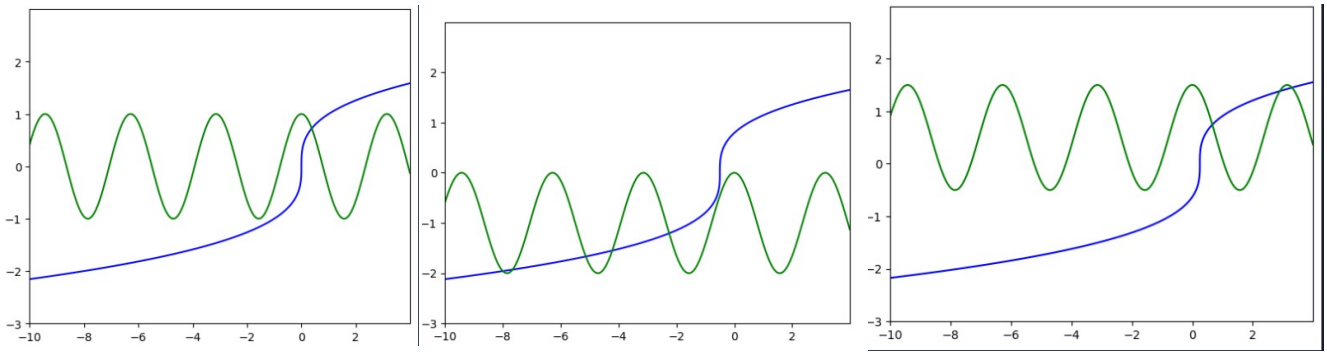
Упрощенный метод Ньютона заключается в том, что мы не пересчитываем Якобиан с новым приближением на каждой итерации, а пользуемся постоянным значением матрицы Якоби для начального приближения:

$$A\Delta x^{(n+1)} = -f(x^{(n)}),$$
$$x^{(n+1)} = x^{(n)} + \Delta x^{(n+1)}.$$

, где $A = f'(x0)$

```
def newton_method_simplified(F, eps, initial):
    x_initial = np.array(initial)
    iteration_number = 0
    jacobian = F.jacobian(X).subs(zip(X, x_initial))
    while np.linalg.norm(np.array(F.subs(zip(X, x_initial)),dtype=float).flatten()) > eps:
        iteration = np.array((jacobian.inv()*F).subs(zip(X, x_initial)), dtype=float).flatten()
        x_initial = x_initial - iteration
        iteration_number += 1
    return x_initial, iteration_number
```

Локализиурем все корни для нашей системы уравнений графически для трех значений параметра a:



И уточним их значения при помощи упрощенного метода Ньютона:

1)
```
print(newton_method_simplified(F2.subs([(a, alpha[0])]), 0.00001, [0.3, 0.7]))
```
Executed at 2024.03.22 02:30:23 in 632ms
```
(array([0.38025858, 0.72448048]), 5)
```

2)
```
print(newton_method_simplified(F2.subs([(a,alpha[1])]), 0.00001, [-2, -1.5]))
print(newton_method_simplified(F2.subs([(a,alpha[1])]), 0.00001, [-4, -1.6]))
print(newton_method_simplified(F2.subs([(a,alpha[1])]), 0.00001, [-5.5, -1.7]))
print(newton_method_simplified(F2.subs([(a,alpha[1])]), 0.00001, [-7.6, -2]))
print(newton_method_simplified(F2.subs([(a,alpha[1])]), 0.00001, [-8, -2]))
```
Executed at 2024.03.22 02:30:25 in 501ms
```
(array([-2.25261906, -1.20567293]), 11)
(array([-4.21765437, -1.54913632]), 6)
(array([-5.13268286, -1.66703306]), 7)
(array([-7.66258537, -1.92762809]), 6)
(array([-8.00034007, -1.95746345]), 4)
```

3)
```
print(newton_method_simplified(F2.subs([(a,alpha[2])]), 0.00001, [0.7, 0.8]))
print(newton_method_simplified(F2.subs([(a,alpha[2])]), 0.00001, [2.4, 1.6]))
print(newton_method_simplified(F2.subs([(a,alpha[2])]), 0.00001, [2.7, 1.7]))
```
Executed at 2024.03.22 02:30:37 in 483ms
```
(array([0.66207542, 0.74415181]), 4)
(array([2.89776215, 1.38343741]), 18)
(array([2.89777867, 1.38344208]), 14)
```

# 5.1. Решение систем линейных уравнений

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5.1.6 | 7.92 | 3.36 | -2.24 | 1.98 | -1.956 | 5.1.21 | 14.556 |
| | -13.86 | 18.20 | 0 | 3.96 | 62.8 | | -100.54 |
| | -2.97 | 0.20 | 4.80 | 0 | -4.16 | | -1.27 |
| | 5.94 | 0 | -10.60 | 16.83 | 48.31 | | -71.31 |

Зададим матрицы A и b:

```
A = np.array([[7.92  , 3.36 , -2.24 , 1.98],
              [-13.86, 18.20, 0     , 3.96 ],
              [-2.97 , 0.20 , 4.80  , 0    ],
              [5.94  , 0    , -10.60, 16.83]])
b = np.array([14.556, -100.54, -1.27, -71.31])
```

Метод Зейделя представляет из себя модификацию метода Якоби (или модификацию метода Простой Итерации). Для его реализации нужно посчитать матрицу B = B1 + B2, где B1 и B2 – строго нижне и верхне диагональные матрицы, а далее итерационно найти вектор приближенного значения (итерационно, потому что текущий шаг использует результаты предыдущего) по формуле:

$$x^{(k+1)} = B_1 x^{(k+1)} + B_2 x^{(k)} + c.$$

Или если в более понятным для программирования виде, то :

$$x_i^{(k+1)} = \sum_{j=1}^{i-1} c_{ij} x_j^{(k+1)} + \sum_{j=i}^{n} c_{ij} x_j^{(k)} + d_i, \quad i = 1, \ldots, n.$$

Преобразуем систему в удобный для итераций вид, найдя матрицу B:

```
B = np.empty(A.shape, dtype=float)
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        B[i, j] = - A[i, j] / A[i, i] if i != j else 0

c = np.empty(b.shape, dtype=float)
for i in range(c.shape[0]):
    c[i] = b[i] / A[i, i]
```

Проверка на ДУ сходимости метода (проверка на сжимающее отображение):

```
print(np.linalg.norm(B, ord=np.inf) < 1)
```

Сам метод Зейделя:

```
def zeid_iterations(B, x0, n):
    answer = x0
    for _ in range(n):
        iteration = np.zeros(answer.shape, dtype=float)
        for i in range(B.shape[0]): # x_new = B1 @ x_new + B2 @ x + c
            iteration[i] = np.sum(B[i][:i] * iteration[:i]) + np.sum(B[i][i:] * answer[i:]) + c[i]
        answer = iteration
    return answer
```

Сделаем 10 итераций этим методом и найдем величину абсолютной погрешностиЮ принимая решение методом Гаусса за точное:

```
True
Initial guess: [0. 0. 0. 0.]
Gauss: x_true = array([ 4.17031201, -1.4316858 ,  2.3754508 , -4.21282679])
Seidel: x_zeid = array([ 4.17031253, -1.43168539,  2.3754511 , -
4.21282678])
Absolute error: 5.203217883220645e-07

Initial guess: [1. 1. 1. 1.]
Gauss: x_true = array([ 4.17031201, -1.4316858 ,  2.3754508 , -4.21282679])
Seidel: x_zeid = array([ 4.17031263, -1.43168531,  2.37545116, -
4.21282678])
Absolute error: 6.25362384987227e-07
```

Алгоритм находит корректные решения системы (совпадают с методом Гаусса), с минимальными различиями, которые обусловленны машинной точностью и числом итераций. Так как условие сходимости выполняется, зависимости от начального приближения почти не наблюдается.

## 5.2.

Модифицируем алгоритм для нахождения решения с точностью epsilon. Для этого воспользуемся известной формулой (которая работает при выполнении ДУ сходимости метода):

$$\|x^{(n)} - x^{(n-1)}\| \|B_2\| / (1 - \|B\|) < \varepsilon.$$

```
def zeid_epsilon(B, x0, eps):
    B1 = np.zeros(B.shape)
    B2 = np.zeros(B.shape)
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            if j < i:
                B1[i, j] = B[i, j]
            if j > i:
                B2[i, j] = B[i, j]
    # B1 - strictly lower triangular matrix
    # B2 - strictly upper triangular matrix

    iterations_number = 0
    answer = x0
    error = 1

    while error > eps:
        x_new = np.zeros(answer.shape, dtype=float)
        for i in range(B.shape[0]): # x_new = B1 @ x_new + B2 @ x + c
            x_new[i] = np.sum(B[i][:i] * x_new[:i]) + np.sum(B[i][i:] * answer[i:]) + c[i]

        # Known formula - stop criteria
        error = np.linalg.norm(answer - x_new)*np.linalg.norm(B2, ord=np.inf) / (1 - np.linalg.norm(B,
ord=np.inf))

        answer = x_new
        iterations_number += 1
    return answer, iterations_number
```

Найдем решение системы выше:

Seidel: x_zeid = (array([ 4.17031201, -1.4316858 ,  2.3754508 , -4.21282679]), 14)

## 5.2. Приложение: Код программы

Построение графиков:

```
x, y = np.meshgrid(np.arange(-2, 2, 0.005), np.arange(-2, 2, 0.005))
plt.figure(figsize=(6, 5))
plt.contour(x, y, np.tan(x*y) - x**2, [0], colors=['blue'])
plt.contour(x, y, 0.7*x**2 + 2*y**2 - 1, [0], colors=['green'])
plt.show()
```

Метод Ньютона:

```
def newton_method(F, eps, initial):
    x_initial = np.array(initial)
    iteration_number = 0
    jacobian = F.jacobian(X)
    while np.linalg.norm(np.array(F.subs(zip(X, x_initial)),dtype=float).flatten()) > eps:
        iteration = np.array(
            (jacobian.inv()*F) #Final equation from the textbook
            .subs(zip(X, x_initial)), dtype=float).flatten()
        x_initial = x_initial - iteration # Iteration
        iteration_number += 1
    return x_initial, iteration_number
```

```
def solve_F1_scipy(x_initial):
    def function(variable: np.array):
        return np.array([np.tan(variable[0]*variable[1]) - variable[0] * variable[0],
                         0.7*variable[0]*variable[0] + 2*variable[1]*variable[1] - 1], dtype=float)
    solution = sp.optimize.fsolve(function, np.array(x_initial), xtol=0.000001)
    return solution
```

```
print(f"Newton's method: {newton_method(F1, 0.000001, [0, -0.5])}; Scipy's fsolve:
{solve_F1_scipy([0, -0.5])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [0, 0.5])}; Scipy's fsolve:
{solve_F1_scipy([0, 0.5])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [0.6, 0.6])}; Scipy's fsolve:
{solve_F1_scipy([0.6, 0.6])}")
print(f"Newton's method: {newton_method(F1, 0.000001, [-0.6, -0.6])}; Scipy's fsolve:
{solve_F1_scipy([-0.6, -0.6])}")
```

Упрощенный метод Ньютона:

```
a = symbols('a')
f21 = x1 - x2**3 + 0.5 * a
f22 = cos(2*x1) - x2 - a

F2 = Matrix([f21, f22])
```

```
def newton_method_simplified(F, eps, initial):
    x_initial = np.array(initial)
    iteration_number = 0
    jacobian = F.jacobian(X).subs(zip(X, x_initial))
    while np.linalg.norm(np.array(F.subs(zip(X, x_initial)),dtype=float).flatten()) > eps:
```

```
        iteration = np.array((jacobian.inv()*F).subs(zip(X, x_initial)), dtype=float).flatten()
        x_initial = x_initial - iteration
        iteration_number += 1
    return x_initial, iteration_number
```

```
print(newton_method_simplified(F2.subs([(a, alpha[0])]), 0.00001, [0.3, 0.7]))
```

Метод Зейделя:

```
A = np.array([[7.92  , 3.36 , -2.24 , 1.98],
              [-13.86, 18.20, 0     , 3.96 ],
              [-2.97 , 0.20 , 4.80  , 0    ],
              [5.94  , 0    , -10.60, 16.83]])
b = np.array([14.556, -100.54, -1.27, -71.31])

B = np.empty(A.shape, dtype=float)
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        B[i, j] = - A[i, j] / A[i, i] if i != j else 0

c = np.empty(b.shape, dtype=float)
for i in range(c.shape[0]):
    c[i] = b[i] / A[i, i]

print(np.linalg.norm(B, ord=np.inf) < 1)


def zeid_iterations(B, x0, n):
    answer = x0
    for _ in range(n):
        iteration = np.zeros(answer.shape, dtype=float)
        for i in range(B.shape[0]): # x_new = B1 @ x_new + B2 @ x + c
            iteration[i] = np.sum(B[i][:i] * iteration[:i]) + np.sum(B[i][i:] * answer[i:]) + c[i]
        answer = iteration
    return answer


initial_values = [np.zeros(b.shape[0]), np.ones(b.shape[0])]
for initial in initial_values:
    x_true = np.linalg.solve(A, b)
    x_zeid = zeid_iterations(B, initial, 10)
    print(f"Initial guess: {initial}")
    print(f"Gauss: {x_true = }")
    print(f"Seidel: {x_zeid = }")

    print(f"Absolute error: {np.linalg.norm(x_true - x_zeid, ord=np.inf)}\n")
```

```
def zeid_epsilon(B, x0, eps):
    B1 = np.zeros(B.shape)
    B2 = np.zeros(B.shape)
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            if j < i:
                B1[i, j] = B[i, j]
            if j > i:
                B2[i, j] = B[i, j]
```

```
    # B1 - strictly lower triangular matrix
    # B2 - strictly upper triangular matrix

    iterations_number = 0
    answer = x0
    error = 1

    while error > eps:
        x_new = np.zeros(answer.shape, dtype=float)
        for i in range(B.shape[0]): # x_new = B1 @ x_new + B2 @ x + c
            x_new[i] = np.sum(B[i][:i] * x_new[:i]) + np.sum(B[i][i:] * answer[i:]) + c[i]

        # Known formula - stop criteria
        error = np.linalg.norm(answer - x_new)*np.linalg.norm(B2, ord=np.inf) / (1 - np.linalg.norm(B,
ord=np.inf))

        answer = x_new
        iterations_number += 1
    return answer, iterations_number

x_zeid = zeid_epsilon(B, np.zeros(b.shape[0]), 0.000001)
print(f"Seidel: {x_zeid = }")
```