МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Лабораторная работа №1
«Численные методы»
Вариант 1

Бобовоза Владислава
Сергеевича
студента 3 курса, 6 группы
специальность «прикладная
математика»

Преподаватель:
Репников В.И.

Минск, 2024

Постановка задачи

1) На примере уравнения $cos\, x = x^3 - 3x + 1$ провести сравнительный анализ следующих методов решения нелинейных уравнений:

   a) метод простой итерации
   b) метод Стеффенсена
   c) метод Ньютона
   d) метод Чебышева третьего порядка

2) Найти с точностью $\varepsilon = 10^{-6}$ наибольший по модулю корень уравнения

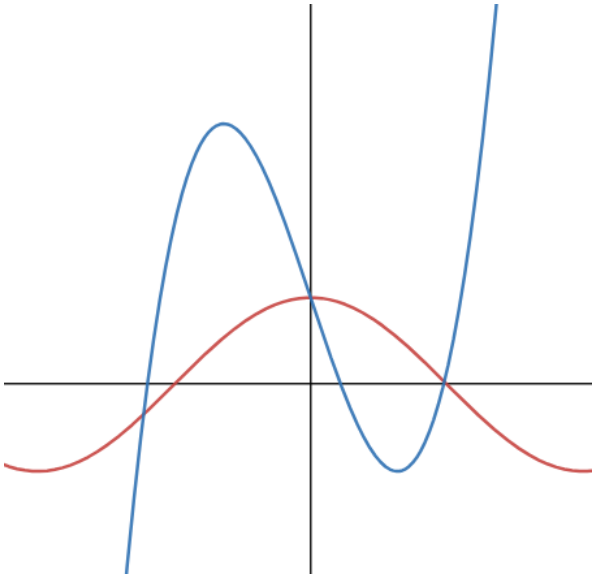
$$\sum_{i=0}^{n} a_i x^i = 0,$$

   где вектор коэффициентов a есть решение системы линейных алгебраических уравнений $Aa = f$

$$A = \begin{pmatrix} 5 & -2 & 0.5 & 1 & -0.1 \\ 1 & -8 & -3.1 & 1 & 2.3 \\ -1 & 3 & 10 & 2 & 4.6 \\ 0 & 0.1 & 2 & 15 & 2 \\ 1 & -2 & 0.4 & 3.2 & -17 \end{pmatrix} \text{ и } f = \begin{pmatrix} 1 \\ 0 \\ 2 \\ -3 \\ 5 \end{pmatrix}.$$

Отделение корней

Будем рассматривать уравнение $cos\, x = x^3 - 3x + 1$.
Построим схематический график, из которого видно, что рассматриваемое уравнение будет иметь 3 корня.



Запишем его в виде $f(x) = 0$, тогда получим $f(x) = cos\, x - x^3 + 3x - 1$.

Отделять корни будем методом дихотомии.
Рассмотрим точки $x = -\pi, x = -\frac{\pi}{2}$.

$$f(-\pi) = cos(-\pi) - (-\pi)^3 + 3(-\pi) - 1 \approx 19.581$$

$$f\left(-\frac{\pi}{2}\right) = cos\left(-\frac{\pi}{2}\right) - \left(-\frac{\pi}{2}\right)^3 + 3\left(-\frac{\pi}{2}\right) - 1 \approx -1.836$$

Значит на отрезке $\left[-\pi, -\frac{\pi}{2}\right]$ существует корень. Уменьшим отрезок:

$f\left(-\frac{3\pi}{4}\right) = cos\left(-\frac{3\pi}{4}\right) - \left(-\frac{3\pi}{4}\right)^3 + 3\left(-\frac{3\pi}{4}\right) - 1 \approx 4.305$, значит $\left[-\frac{3\pi}{4}, -\frac{\pi}{2}\right]$.

$f\left(-\frac{5\pi}{8}\right) = cos\left(-\frac{5\pi}{8}\right) - \left(-\frac{5\pi}{8}\right)^3 + 3\left(-\frac{5\pi}{8}\right) - 1 \approx 0.296$, значит $\left[-\frac{5\pi}{8}, -\frac{\pi}{2}\right]$.

Проверим единственность корня на полученном отрезке:

$$f'(x) = -\sin x - 3x^2 + 3$$

$$f'(x) < 0, \forall x \in \left[-\frac{5\pi}{8}, -\frac{\pi}{2}\right].$$

Отсюда следует, что на $\left[-\frac{5\pi}{8}, -\frac{\pi}{2}\right]$ существует единственный корень уравнения $f(x) = 0$.

Метод простой итерации

Для применения метода простой итерации в первую очередь необходимо привести уравнение к виду $x = \varphi(x)$.

Теорема о сходимости метода простой итерации.
Пусть выполняются следующие условия:

1) функция $\varphi(x)$ определена на отрезке $|x - x^0| \leq \delta$, непрерывна на нем и всех x из отрезка $|x - x^0| \leq \delta$ функция $\varphi(x)$ имеет непрерывную первую производную $|\varphi'(x)| < q$ для любого $x \in [x^0 - \delta, x^0 + \delta]$
2) для начального приближения x^0 верно неравенство $|x^0 - \varphi(x^0)| \leq m$
3) числа δ, q, m удовлетворяют условию

$$\frac{m}{1-q} \leq \delta$$

Тогда:

1) уравнение $x = \varphi(x)$ в области $|x - x^0| \leq \delta$ имеет решение
2) последовательность x^k построенная по правилу $x^{k+1} = \varphi(x^k)$ принадлежит отрезку $[x^0 - \delta, x^0 + \delta]$, является $\lim\limits_{k\to\infty} x^k = x^*$
3) скорость сходимости x^k к x^* оценивается неравенством $\left|x^* - x^k\right| \leq \frac{m}{1-q} q^k$

Канонический вид уравнения имеет следующий вид:

$$\varphi(x) = x + \frac{1}{7.64206}(\cos x - x^3 + 3x - 1)$$

В качестве начального приближения x^0 выберем левую границу $x^0 = -\frac{5\pi}{8}$.

Листинг программы на языке Python:

```
from math import pi, cos

x0 = -5*pi/8
eps = 1e-8
iter_counter = 0

while True:
    xk = x0 + 1/7.64206 * (cos(x0) - x0**3 + 3*x0 - 1)
    iter_counter += 1
    print(f"Итерация: {iter_counter}. Значение: {xk}")
    if abs(x0 - xk) < eps:
        break
    else:
        x0 = xk
```

Результаты выполнения программы:

```
Итерация: 1. Значение: -1.9246679052019204
Итерация: 2. Значение: -1.923476982035899
Итерация: 3. Значение: -1.9234030930823471
Итерация: 4. Значение: -1.923398436063382
Итерация: 5. Значение: -1.9233981422597322
Итерация: 6. Значение: -1.923398123723013
Итерация: 7. Значение: -1.9233981225534862
```

Метод Стеффенсена

Для метода последовательных приближений существует модификация, которая позволяет ускорить процесс. Она основана на преобразовании Эйткена.

Итогом преобразования является следующий итерационный процесс:

$$x^{k+1} = \frac{x^k \varphi\left(\varphi(x^k)\right) - \left(\varphi(x^k)\right)^2}{\varphi\left(\varphi(x^k)\right) - 2\varphi(x^k) + x^k}, \qquad k = 0,1, \dots, x^0$$

Преобразуем полученный итерационный процесс, тогда:

$$x^{k+1} = x^k - \frac{(\varphi(x^k) - x^k)^2}{\varphi\left(\varphi(x^k)\right) - 2\varphi(x^k) + x^k}.$$

Листинг программы на языке Python:

```
from math import pi, cos


def phi(x):
    return x + 1/7.64206 * (cos(x) - x**3 + 3*x - 1)

x0 = -5*pi/8
eps = 1e-8
iter_counter = 0

while True:
    xk = x0 - (phi(x0) - x0) ** 2 / (phi(phi(x0)) - 2 * phi(x0) + x0)
    iter_counter += 1
    print(f"Итерация: {iter_counter}. Значение: {xk}")
    if abs(x0 - xk) < eps:
        break
    else:
        x0 = xk
```

Результаты выполнения программы:

```
Итерация: 1. Значение: -1.9234392980066213
Итерация: 2. Значение: -1.9233981225634673
Итерация: 3. Значение: -1.9233981224747287
```

Метод Ньютона

Методом Ньютона называется следующий итерационный процесс:

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}, \qquad k = 0,1,\dots, x^0$$

Листинг программы на Python:

```
from math import pi, sin, cos


def f(x):
    return cos(x) - x**3 + 3*x - 1

def diff_f(x):
    return -sin(x) - 3*x**2 + 3

x0 = -5*pi/8
eps = 1e-8
iter_counter = 0

while True:
    xk = x0 - f(x0)/diff_f(x0)
    iter_counter += 1
    print(f"Итерация: {iter_counter}. Значение: {xk}")
    if abs(x0 - xk) < eps:
        break
    else:
        x0 = xk
```

Результаты выполнения программы:

```
Итерация: 1. Значение: -1.9246679210793791
Итерация: 2. Значение: -1.92339945863356
Итерация: 3. Значение: -1.9233981224762107
Итерация: 4. Значение: -1.9233981224747287
```

Метод Чебышева третьего порядка

Методом Чебышева третьего порядка называется итерационный процесс вида:

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} - \frac{f^2(x^k)f''(x^k)}{2(f'(x^k))^3}.$$

Метод имеет кубическую скорость сходимости.

Листинг программы на Python:

```
from math import pi, sin, cos


def f(x):
    return cos(x) - x**3 + 3*x - 1

def diff_f(x):
    return -sin(x) - 3*x**2 + 3

def double_diff_f(x):
    return -cos(x) - 6*x

x0 = -5*pi/8
eps = 1e-8
iter_counter = 0

while True:
    xk = x0 - f(x0)/diff_f(x0) - (f(x0))**2 * double_diff_f(x0) / (2 *
(diff_f(x0))**3)
    iter_counter += 1
    print(f"Итерация: {iter_counter}. Значение: {xk}")
    if abs(x0 - xk) < eps:
        break
    else:
        x0 = xk
```

Результаты выполнения программы:

```
Итерация: 1. Значение: -1.923468139792224
Итерация: 2. Значение: -1.9233981224751462
Итерация: 3. Значение: -1.9233981224747287
```

Вывод относительно методов решения нелинейных уравнений

Если сравнить полученные результаты и средства, которыми они достигались, можно сделать следующий вывод.

Метод Стеффенсена и метод Чебышева третьего порядка позволяют получить результат с необходимой точностью за наименьшее количество итераций. В свою очередь, для метода Стеффенсена необходимо вычислить $\varphi(x)$, что требует предварительной подготовки и дополнительных затрат. Для метода Чебышева необходимо вычислить первую и вторую производные.

Для использования метода Ньютона, который на мой взгляд является самым сбалансированным, необходимо проверить, выполняются ли условия теоремы. Если они не выполняются, то следующим шагом я бы реализовал метод Чебышева третьего порядка.

Метод простой итерации значительно проигрывает остальным методам, как в предварительной подготовке, так и в скорости сходимости.

Решение СЛАУ и нахождение корней полинома

Для начала необходимо решить СЛАУ, чтобы получить коэффициенты a_i для

$$\sum_{i=0}^{n} a_i x^i = 0.$$

Решим систему используя метод Гаусса-Зейделя.

Листинг программы на Python

```
import numpy as np


def transpose_matrix(A_):
    size = A_.shape[0]
    transposed_matrix = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            transposed_matrix[j, i] = A_[i, j]
    return transposed_matrix

def is_symmetric(A_):
    size = A_.shape[0]
    for i in range(size):
        for j in range(size):
            if A_[i][j] != A_[j][i]:
                return False
    return True

def gauss_transformation(A_, b_):
    if is_symmetric(A_):
        return A_, b_
    else:
        return np.dot(transpose_matrix(A_), A_),
np.dot(transpose_matrix(A_), b_)

def gauss_seidel(A_, b_, max_iterations=1000, tolerance=1e-6):
    size = A_.shape[0]
    A_, b_ = gauss_transformation(A_, b_)

    x_ = np.zeros(size)
    for k in range(max_iterations):
        x_new = np.copy(x_)
        for i in range(size):
            sigma = np.dot(A_[i, :i], x_new[:i]) + np.dot(A_[i, i + 1:],
x_[i + 1:])
            x_new[i] = (1 / A_[i, i]) * (b_[i] - sigma)

        if np.linalg.norm(x_new - x_) < tolerance:
            return x_new

        x_ = x_new
```

```
    raise ValueError("Метод релаксации не сошелся за максимальное число
итераций")


if __name__ == '__main__':
    # Инициализация матрицы A
    A = np.array([[5, -2, 0.5, 1, -0.1],
                  [1, -8, -3.1, 1, 2.3],
                  [-1, 3, 10, 2, 4.6],
                  [0, 0.1, 2, 15, 2],
                  [1, -2, 0.4, 3.2, -17]])

    # Инициализация вектора f
    b = np.array([1, 0, 2, -3, 5])

    # Получение коэффициентов путем решения СЛАУ
    a = gauss_seidel(A, b).tolist()

    # Вывод полинома:
    poly = ""
    for i in range(5):
        poly += f"({a[4-i]} * x^{4-i}) + "
    poly = poly[:-3]

    print(f"После решения СЛАУ, получим коэффициенты, из которых получим
многочлен следующего вида:\n{poly}")
```

Результаты выполнения программы

```
После решения СЛАУ, получим коэффициенты, из которых получим многочлен
следующего вида:
(-0.2870922787947789 * x^4) + (-0.22238701134031116 * x^3) +
(0.4691074958043837 * x^2) + (-0.282250798823043 * x^1) +
(0.07892474533288997 * x^0)
```

Как итог, мы получили полином, корни которого необходимо найти с точностью $\varepsilon = 10^{-6}$.

Искать его корни будет с помощью схемы Горнера.

Листинг функций на Python:

```
def f_x0(polynom, x0):
    value = 0
    for i in range(len(polynom)):
        value += polynom[i] * (x0**i)
    return value

def diff_f_x0(polynom, x0):
    new_poly = []
    for i in range(1, len(polynom)):
        el_poly = polynom[i] * i * (x0**(i-1))
        new_poly.append(el_poly)
    return sum(new_poly)
```

```
def newton(polynom, x0, eps=1e-6):
    iter_counter = 0
    while True:
        xk = x0 - f_x0(polynom, x0) / diff_f_x0(polynom, x0)
        iter_counter += 1
        if abs(x0 - xk) >= eps:
            x0 = xk
        else:
            break
    return xk

def horner_scheme(poly, x0):
    n = len(poly)
    result = [0] * (n - 1)  # Инициализируем новый полином нулями

    # Проходим по коэффициентам, начиная с последнего
    for i in range(n - 1, 0, -1):
        result[i - 1] = poly[i] + (result[i] if i < n - 1 else 0) * x0

    return result

def roots_of_poly(polynom, x0=0):
    roots = []
    while True:
        if len(polynom) == 3:
            # Если у нас осталось квадратное уравнение
            a, b, c = polynom
            D = b ** 2 - 4 * a * c
            if D < 0:
                # Корней нет
                break
            else:
                root1 = (-b + D ** 0.5) / (2 * a)
                root2 = (-b - D ** 0.5) / (2 * a)
                print("D, ", root1, root2)
                roots.append(root1)
                roots.append(root2)
                break
        else:
            root = newton(polynom, x0)
            roots.append(root)
            polynom = horner_scheme(polynom, root)
    return roots
```

Вызовем функцию roots_of_poly(a).

Результаты выполнения программы:

```
Корни полученного многочлена: [0.5640552356683952, -1.9262924858939339]
Максимальный по модулю корень: 1.9262924858939339
```

Листинг полной программы:

```
import numpy as np


def transpose_matrix(A_):
    size = A_.shape[0]
    transposed_matrix = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            transposed_matrix[j, i] = A_[i, j]
    return transposed_matrix

def is_symmetric(A_):
    size = A_.shape[0]
    for i in range(size):
        for j in range(size):
            if A_[i][j] != A_[j][i]:
                return False
    return True

def gauss_transformation(A_, b_):
    if is_symmetric(A_):
        return A_, b_
    else:
        return np.dot(transpose_matrix(A_), A_),
np.dot(transpose_matrix(A_), b_)

def gauss_seidel(A_, b_, max_iterations=1000, tolerance=1e-6):
    size = A_.shape[0]
    A_, b_ = gauss_transformation(A_, b_)

    x_ = np.zeros(size)
    for k in range(max_iterations):
        x_new = np.copy(x_)
        for i in range(size):
            sigma = np.dot(A_[i, :i], x_new[:i]) + np.dot(A_[i, i + 1:],
x_[i + 1:])
            x_new[i] = (1 / A_[i, i]) * (b_[i] - sigma)

        if np.linalg.norm(x_new - x_) < tolerance:
            return x_new

        x_ = x_new

    raise ValueError("Метод релаксации не сошелся за максимальное число
итераций")

def f_x0(polynom, x0):
    value = 0
    for i in range(len(polynom)):
        value += polynom[i] * (x0**i)
    return value

def diff_f_x0(polynom, x0):
    new_poly = []
    for i in range(1, len(polynom)):
```

```
        el_poly = polynom[i] * i * (x0**(i-1))
        new_poly.append(el_poly)
    return sum(new_poly)

def newton(polynom, x0, eps=1e-6):
    iter_counter = 0
    while True:
        xk = x0 - f_x0(polynom, x0) / diff_f_x0(polynom, x0)
        iter_counter += 1
        if abs(x0 - xk) >= eps:
            x0 = xk
        else:
            break
    return xk

def horner_scheme(poly, x0):
    n = len(poly)
    result = [0] * (n - 1)  # Инициализируем новый полином нулями

    # Проходим по коэффициентам, начиная с последнего
    for i in range(n - 1, 0, -1):
        result[i - 1] = poly[i] + (result[i] if i < n - 1 else 0) * x0

    return result

def roots_of_poly(polynom, x0=0):
    roots = []
    while True:
        if len(polynom) == 3:
            # Если у нас осталось квадратное уравнение
            a, b, c = polynom
            D = b ** 2 - 4 * a * c
            if D < 0:
                # Корней нет
                break
            else:
                root1 = (-b + D ** 0.5) / (2 * a)
                root2 = (-b - D ** 0.5) / (2 * a)
                print("D, ", root1, root2)
                roots.append(root1)
                roots.append(root2)
                break
        else:
            root = newton(polynom, x0)
            roots.append(root)
            polynom = horner_scheme(polynom, root)
    return roots


if __name__ == '__main__':
    # Инициализация матрицы A
    A = np.array([[5, -2, 0.5, 1, -0.1],
                  [1, -8, -3.1, 1, 2.3],
                  [-1, 3, 10, 2, 4.6],
                  [0, 0.1, 2, 15, 2],
                  [1, -2, 0.4, 3.2, -17]])

    # Инициализация вектора f
```

```
    b = np.array([1, 0, 2, -3, 5])

    # Получение коэффициентов путем решения СЛАУ
    a = gauss_seidel(A, b).tolist()

    # Вывод полинома:
    poly = ""
    for i in range(5):
        poly += f"({a[4-i]} * x^{4-i}) + "
    poly = poly[:-3]
    print(f"После решения СЛАУ, получим коэффициенты, из которых получим
многочлен следующего вида:\n{poly}")

    # Решение многочлена:
    roots = roots_of_poly(a)
    print(f"Корни полученного многочлена: {roots}")
    print(f"Максимальный по модулю корень: {np.max(np.abs(roots))}")
```