Университет ИТМО

Факультет: ПИиКТ

Дисциплина: Вычислительная математика

Лабораторная работа №2

Вариант “2вг”

Выполнили: **Кудлаков Роман**

Группа: **P3231**

Преподаватель: **Перл Ольга Вячеславовна**

Санкт-Петербург, 2021

Метод простой итерации, называемый также **методом** последовательного приближения, – это математический алгоритм нахождения значения неизвестной величины путем постепенного ее уточнения.
На первом шаге сами определяем интервал, на котором будем искать решения.
Далее задаем точность вычислений.
После приводим систему к нормализованному виду

$$X = \Phi(X)\text{, где } \Phi(X) = \begin{pmatrix} \varphi_1(x_1, x_2, \ldots, x_n) \\ \varphi_2(x_1, x_2, \ldots, x_n) \\ \varphi_3(x_1, x_2, \ldots, x_n) \end{pmatrix}$$

$$\varphi_i = x_i + f_i, i = 1,2,\ldots n.$$

И выбираем начальное приближение, где значение каждой неизвестной лежит в своем интервале.
Теперь, чтобы воспользоваться методом нужно проверить выполняется ли условие сходимости на выбранном интервале:

$$\sum_{i=1}^{n} \left| \frac{\partial \varphi_i}{\partial x_i} \right| < 1,\ \text{для } i = 1,2,\ldots,n.\ .$$

Теперь подставим в правую часть каждого уравнения значение из начального приближения и получим новые значения неизвестных. Дальше будем подставлять уже новые, только что полученные значения, в уравнения.

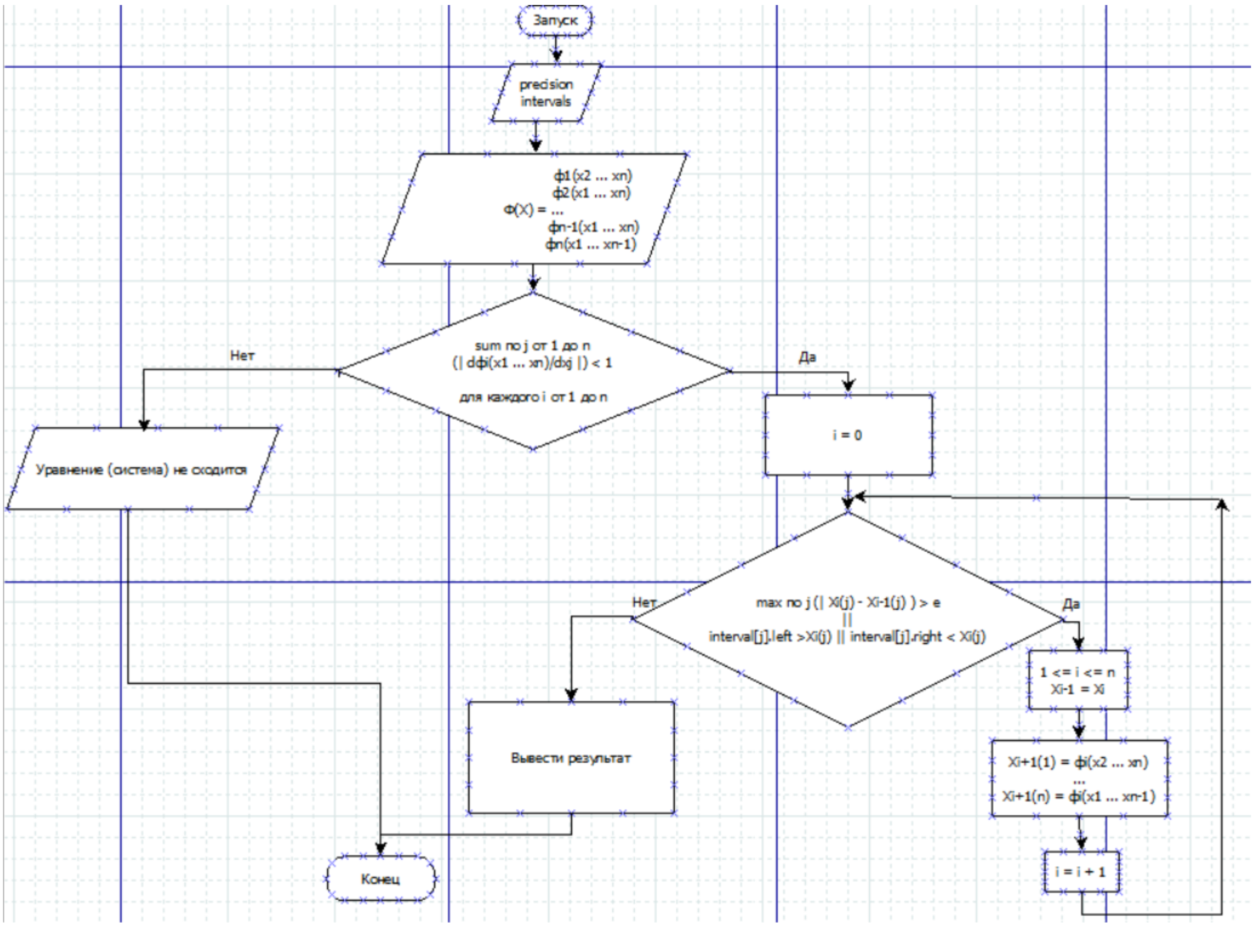
$$X^{(k+1)} = \Phi(X^{(k)}).$$

Так будем продолжать делать до тех пор, пока не получим точность меньше, чем данная.

Расчет точности на i-том шаге:

$$max \sum_{i=1}^{n} |x_i \ - \ x_{i-1}|$$

Блок-схема алгоритма



Метод касательных

Метод предназначен для приближенного нахождения нулей функции. Суть метода касательных состоит в разбиении отрезка [a; b] (при условии f(a) * f(b) < 0) на два отрезка с помощью касательной и выборе нового отрезка от точки пересечения касательной с осью абсцисс до неподвижной точки, на котором функция меняет знак и содержит решение, причём подвижная точка приближается к ε-окрестности решения.

Метод касательных применим для решения уравнения вида **f(x) = 0** на отрезке **[a; b]**

Условие неподвижной точки для метода касательных **f(x) * f''(x) < 0**.

Условие начальной точки для метода касательных **f(x) * f''(x) > 0**.

Сначала выбираем отрезок **[a; b]** такой, что функция **f(x)** дважды непрерывно дифференцируема и меняет знак на отрезке, то есть **f(a) * f(b) < 0**

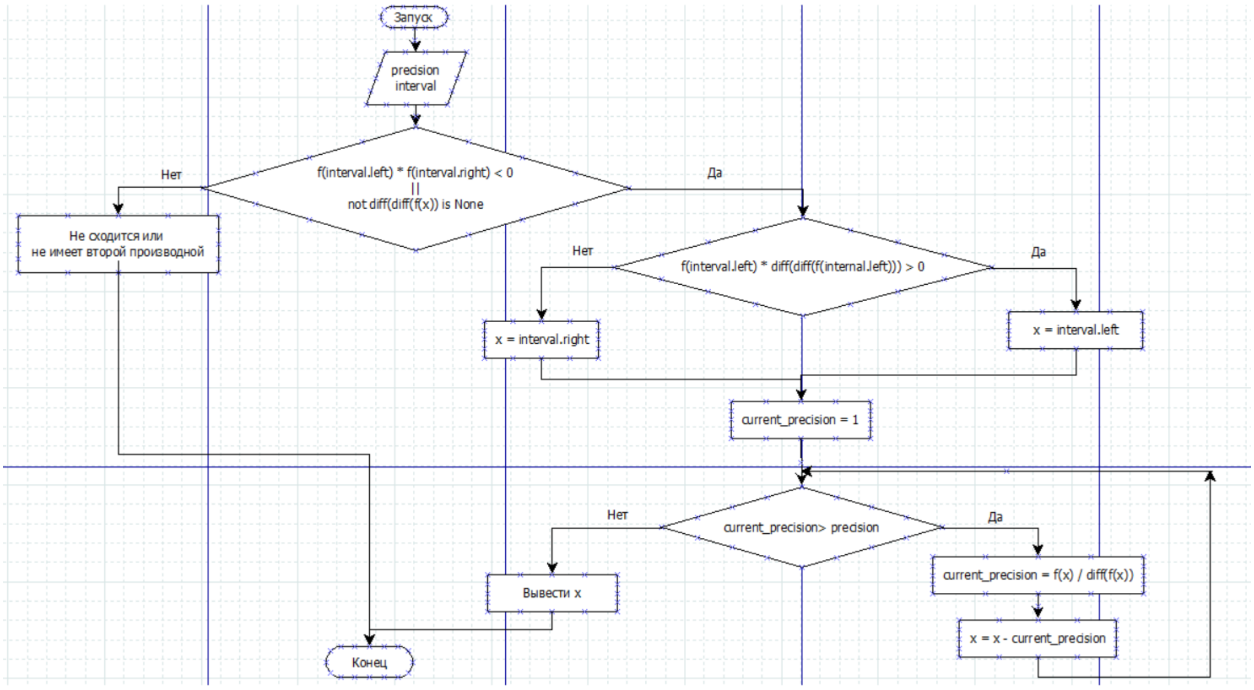
1. Находим начальную точку, если **f(a) * f''(a) > 0**, то **x = a**, иначе если **f(b) * f''(b) > 0**, то **x = b**

2. Дальше воспользуемся формулой нахождения следующей точки x

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Если $\left|\frac{f(x_i)}{f'(x_i)}\right| > \varepsilon$, тогда делаем шаг два заново, иначе выводим результат.

Блок-схема алгоритма



Листинг кода

```
def express_unknowns(function, variable):
    return sp.solve(sp.parse_expr(function), variable)


def make_expressed_functions(funcs, name_vars):
    ex_functions = []
    for i in range(len(funcs)):
        ex_functions.append(express_unknowns(funcs[i], name_vars[i])[0])
    return ex_functions


def take_derivative(function, variable):
    return sp.diff(function, variable)


def find_derivatives(funcs, name_vars):
    diffs = []
    for i in range(len(funcs)):
        row_of_diffs = []
        for j in range(len(funcs)):
            row_of_diffs.append(take_derivative(funcs[i], name_vars[j]))
        diffs.append(row_of_diffs)
    return diffs


def find_critical_points(diffs, name_vars):
    second_diffs = []
    for i in range(len(diffs)):
        row_of_diffs = []
        for j in range(len(diffs[i])):
```

```
            row_of_diffs.append(take_derivative(diffs[i][j], name_vars[j]))
        second_diffs.append(row_of_diffs)

    crit_points = []
    for i in range(len(second_diffs)):
        row_of_crit_points = []
        for j in range(len(second_diffs[i])):
            row_of_crit_points.append(sp.solve(second_diffs[i][j], name_vars[j]))
        crit_points.append(row_of_crit_points)
    return crit_points


def find_max_abs_on_the_interval(crit_points_func, function, variable, interval):
    maximum = 0
    for crit_point in crit_points_func:
        if interval[0] <= crit_point <= interval[1]:
            maximum = max(maximum, abs(function.subs(variable, crit_point)))

    left_border = abs(function.subs(variable, interval[0]))
    right_border = abs(function.subs(variable, interval[1]))
    maximum_between_borders = max(left_border, right_border)
    return max(maximum, maximum_between_borders)


def is_converging(crit_points, diffs, name_vars, intervals):
    for i in range(len(crit_points)):
        sum = 0
        for j in range(len(crit_points[i])):
            sum += find_max_abs_on_the_interval(crit_points[i][j],
derivatives[i][j], variables[j], intervals[j])
        if sum >= 1:
            return False
    return True


def do_method_of_simple_iterations(ex_functions, intervals, precision, name_vars):
    result = {'unknowns': [], 'precisions': [], 'iterations': 0}
    current_step = [0.0, 0.0]
    previous_step = [np.mean(intervals[0]), np.mean(intervals[1])]
    precisions = [0.0, 0.0]
    current_precision = 1
    iteration = 0
    while current_precision >= pow(0.1, precision):
        for i in range(len(previous_step)):
            if intervals[i][0] > previous_step[i] or intervals[i][1] <
previous_step[i]:
                return result

        for i in range(len(ex_functions)):
            current_step[i] = (ex_functions[i].subs([(name_vars[0],
previous_step[0]), (name_vars[1], previous_step[1])])).evalf()

        for i in range(len(current_step)):
            precisions[i] = abs(current_step[i] - previous_step[i])

        current_precision = max(precisions)

        for i in range(len(current_step)):
            previous_step[i] = current_step[i]
        iteration += 1

    result['unknowns'] = previous_step
```

```
    result['precisions'] = precisions
    result['iterations'] = iteration
    return result


def is_converge_by_curve(left_limit, right_limit, func):
    return func.subs('x', left_limit).evalf() * func.subs('x',
right_limit).evalf() < 0



def is_start_point(x, func):
    return func.subs('x', x).evalf() * take_derivative(take_derivative(func, 'x'),
'x').subs('x', x).evalf() > 0



def do_iterations(current_point, func, precision):
    coef = 1
    iteration = 0
    while abs(coef) > 0.1 ** precision:
        coef = func.subs('x', current_point).evalf() / take_derivative(func,
'x').subs('x', current_point).evalf()
        current_point -= coef
        iteration += 1

    accuracy = coef
    return {'iterations': iteration, 'result': current_point, 'accuracy':
accuracy}



def do_method_of_the_curve(left_limit, right_limit, func, precision):
    if not is_converge_by_curve(left_limit, right_limit, func):
        return {'error': "Doesn't converge by method of the curve"}
    start_point = oo
    if is_start_point(left_limit, func):
        start_point = left_limit
    if start_point == oo and is_start_point(right_limit, func):
        start_point = right_limit
    if start_point != oo:
        return do_iterations(start_point, func, precision)
    else:
        return {'error': "No start point"}
```

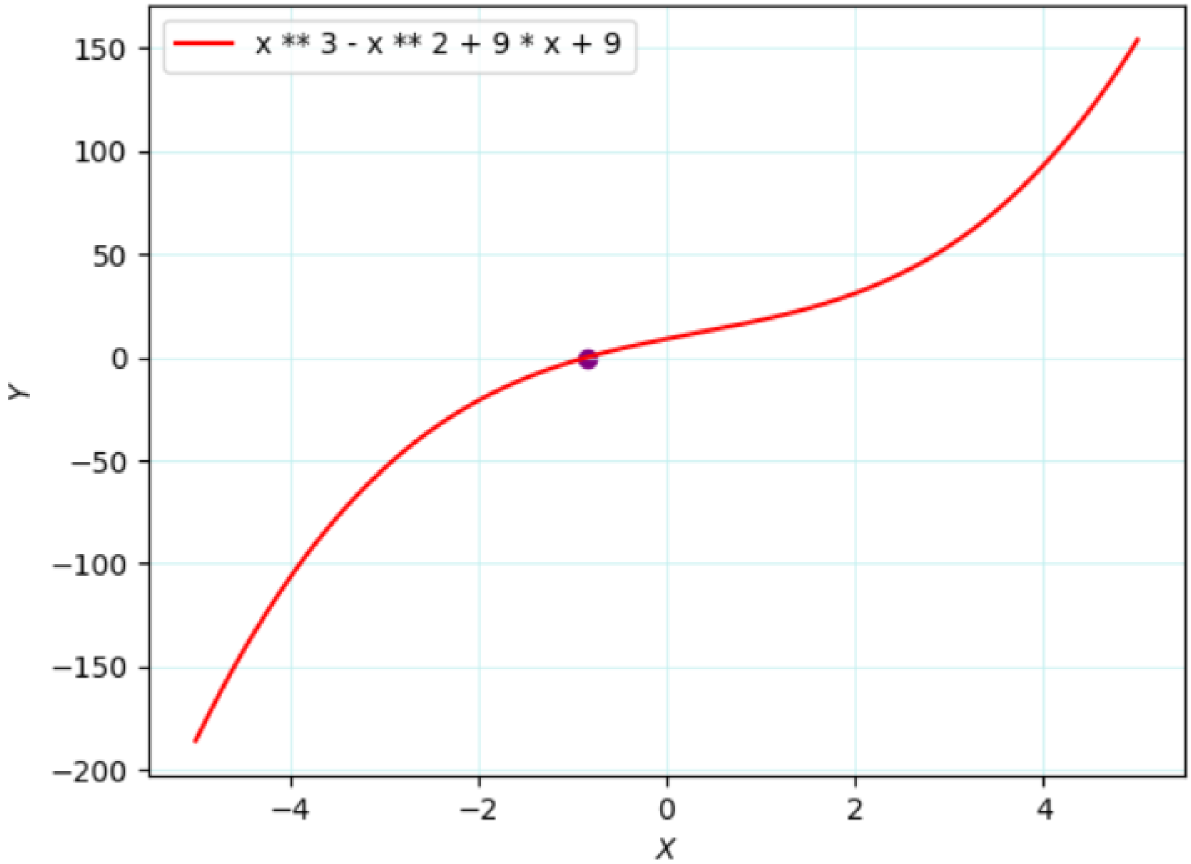
```python
def find_critical_points(func, name_var):
    diff = take_derivative(func, name_var)

    crit_points = [x for x in sp.solve(diff, name_var) if x.is_real]
    return crit_points


def count_coefficient(left_limit, right_limit, func):
    diff = take_derivative(func, 'x')
    crit_points = find_critical_points(diff, 'x')

    max_value = max(abs(diff.subs('x', left_limit).evalf()), abs(diff.subs('x', 
right_limit).evalf()))
    for crit_point in crit_points:
        if left_limit <= crit_point <= right_limit:
            max_value = max(max_value, abs(func.subs('x', crit_point).evalf()))

    return max_value


def is_converge_by_simple_iterations(left_limit, right_limit, func):
    max_value = count_coefficient(left_limit, right_limit, func)

    if max_value >= 1:
        return False

    mid = (right_limit + left_limit) / 2
    return abs(func.subs('x', mid).evalf() - mid) < (1 - max_value) * (mid - 
left_limit)


def do_iterations_method_of_simple_iterations(current_point, func, precision):
    prev_point = current_point
    current_precision = 1
    iteration = 0
    while current_precision > 0.1 ** precision:
```

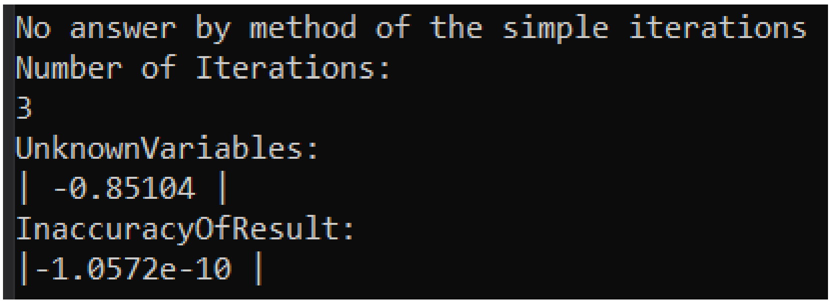
```
        current_point = func.subs('x', prev_point).evalf()
        current_precision = abs(current_point - prev_point)
        prev_point = current_point
        iteration += 1

    return {'iterations': iteration, 'result': current_point, 'accuracy': current_precision}


def do_method_of_simple_iterations_equations(left_limit, right_limit, func, precision):
    if not is_converge_by_simple_iterations(left_limit, right_limit, func):
        return {'error': "Doesn't converge by method of simple iterations"}

    answer = do_iterations_method_of_simple_iterations((left_limit + right_limit) / 2, func, precision)
    coef = count_coefficient(left_limit, right_limit, func)
    answer['coef'] = coef
    return answer
```

Тесты и результаты

Тест 1.

x^3 - x^2 + 9x + 9 = 0

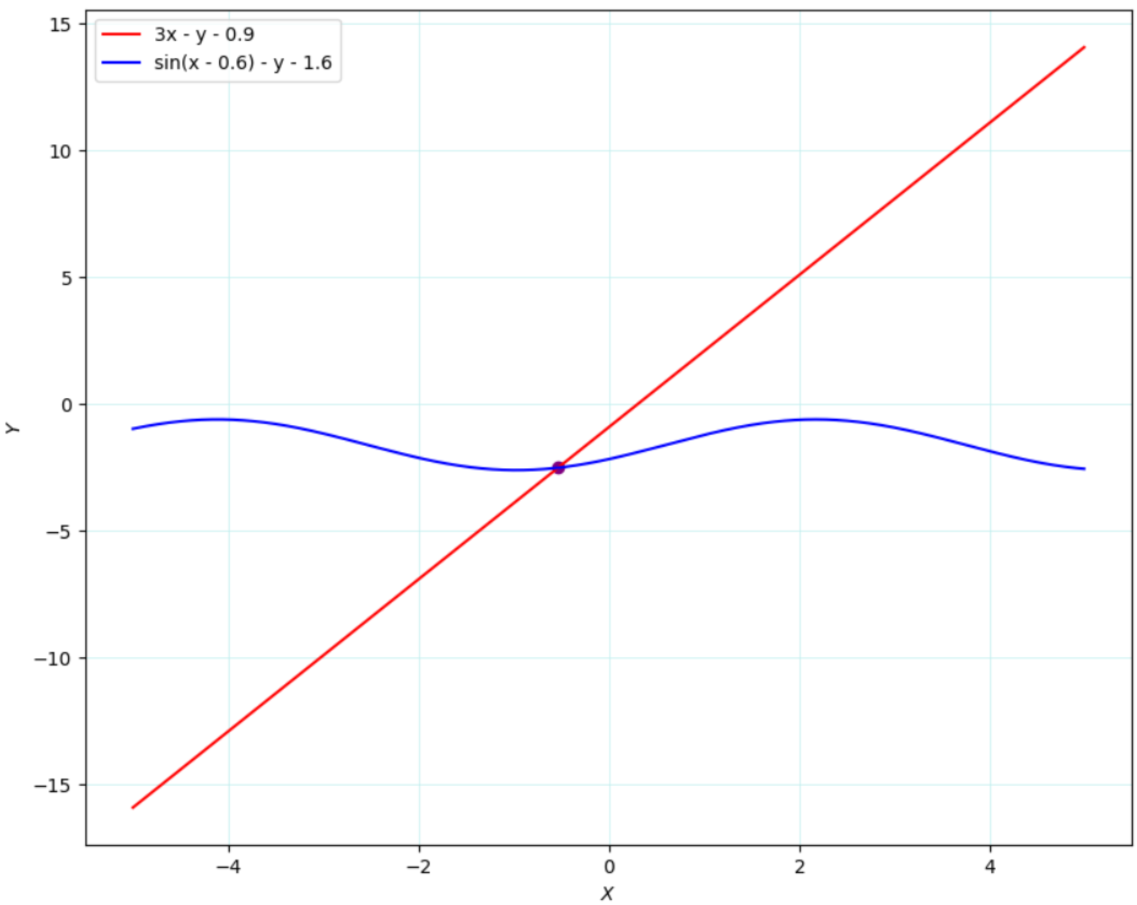




```
No answer by method of the simple iterations
Number of Iterations:
3
UnknownVariables:
| -0.85104 |
InaccuracyOfResult:
|-1.0572e-10 |
```

Тест 2.

$$\begin{cases} 3x - y = 0.9 \\ \sin(x - 0.6) - y = 1.6 \end{cases}$$




```
Result:
Number of iterations: 7
x = -0.535588776052434 +- 0.0000415826
y = -2.50703284379524 +- 0.0000186561
```

Вывод.

Метод касательных имеет высокую скорость сходимости, так как имеет квадратичную сходимость, однако, чтобы воспользоваться этим методом функция должна обязательно:

1. Иметь вторую производную

2. Первая производная не равна 0

3. Знакопостоянство первой и второй производных

Поэтому желательно использовать данный метод с другими методами и выбирать как можно меньший интервал поиска решения.

Метод простой итерации может использоваться не на очень большом количестве уравнений, так как нужно, чтобы выполнялся признак сходимости и решение было единственно на отрезке.