1. Описание метода. Расчётные формулы.

Метод деления пополам

Может найти корень функции на отрезке $[a, b]$, если $f(a)f(b) < 0$ (т.е. знаки функций на концах отрезка противоположны). Шаг метода:

- найти середину отрезка
- проверить, является ли она корнем (если да, то завершаем)
- выбрать ту половину, для которой $f(a_1)f(b_1) < 0$

Таким образом длинна отрезка: $l_k = (b_0 - a_0)2^{-k}$ что при $k \to \infty$ обратится в конкретную точку-корень

Метод простой итерации

Если превратить уравнение $f(x) = 0$ в $x = g(x)$, то можно считать точное значение x через его приближённые значения, причём начать можно с любого числа из ОДЗ. Таким образом $x_{k+1} = g(x_k)$, и при $k \to \infty$ можно найти точное значение корня.

Метод Ньютона для систем уравнений

Пусть система уравнений задана как:

$$\begin{cases} F_1(x_1, ..., x_n) = 0 \\ .... \\ F_n(x_1, ..., x_n) = 0 \end{cases} \implies F = (F_1, ...F_n), X = (x_1, ..., x_n)$$

Её решение по методу Ньютона находится так:

$$X^{(k+1)} = X^{(k)} - \Delta X^{(k)} = X^{(k)} - J^{-1}\left(X^{(k)}\right) F\left(X^{(k)}\right)$$

$$J = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & ... & \frac{\partial f_1(x)}{\partial x_n} \\ ... & ... & ... \\ \frac{\partial f_n(x)}{\partial x_1} & ... & \frac{\partial f_n(x)}{\partial x_n} \end{pmatrix}$$

Необходимо также определить начальное приближение корней $\Delta X^{(0)}$

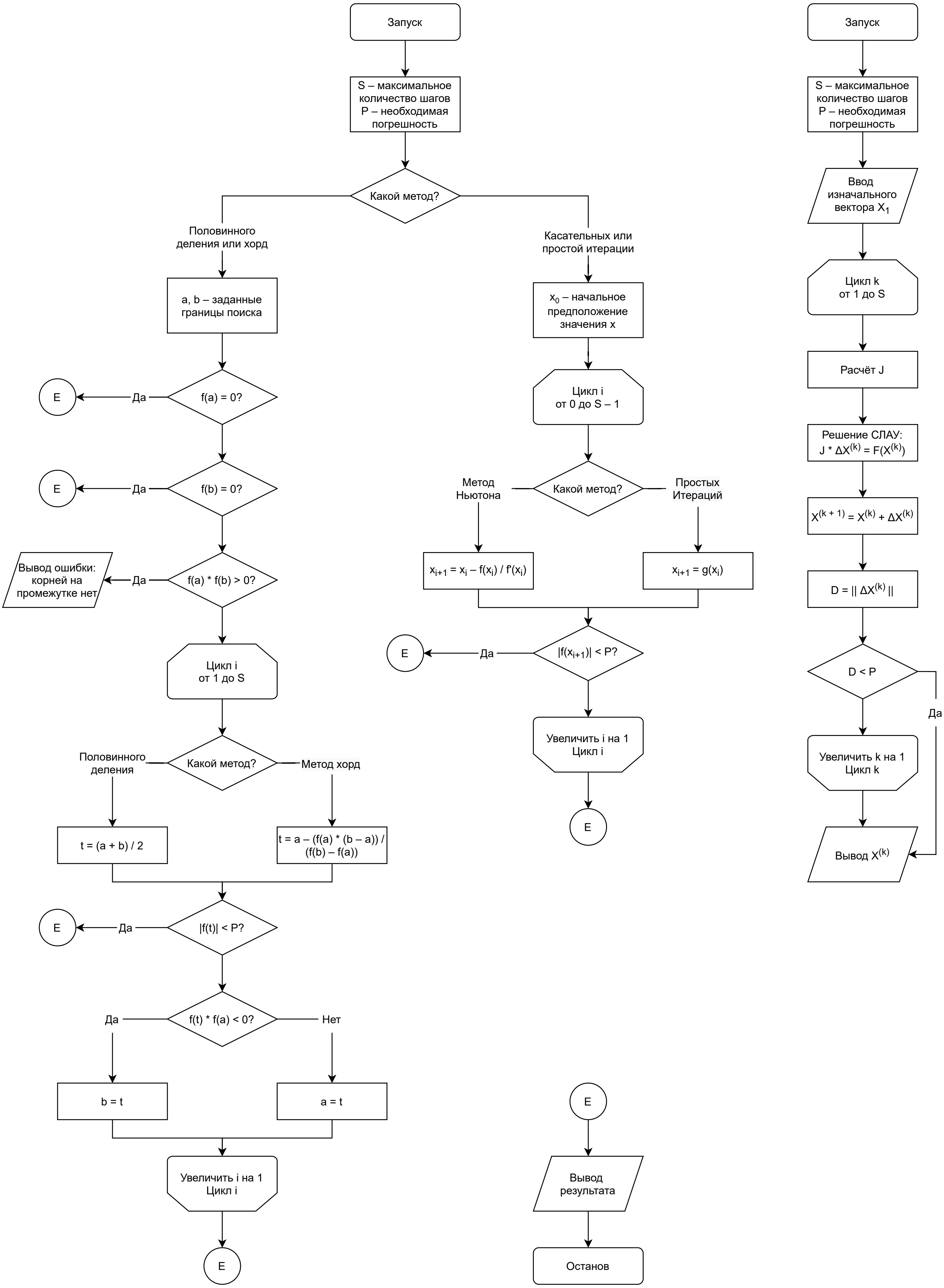
Таким образом на каждом шаге нужно находить обратную J матрицу, что проще делать через решение СЛАУ:

$$J\left(X^{(k)}\right) \Delta X^{(k)} = F\left(X^{(k)}\right)$$

Выполнение метода можно считать завершённым, как только погрешность, $||\Delta X^{(k)}||$ становится меньше необходимой точности.

2. Блок-схема

Схема слева — для решения уравнений, с права — для решения систем уравнений



3. Листинг численного метода

Уравнения

```
class ParamSpec:
    def convert(self) -> ...:
        raise NotImplementedError()


class Solver:
    def __init__(self, root_precision: int = None):
        self.precision: Decimal = Decimal(f"1E-{root_precision or 20}")

    def solve(self, equation: AnyEquation, params: ParamSpec) -> Decimal:
        raise NotImplementedError()


class DifferentialSolver(Solver, ABC):
    def __init__(self, max_steps: int = None, root_precision: int = None):
        super().__init__(root_precision)
        self.max_steps: int | None = max_steps

    def is_root(self, y: Decimal) -> bool:
        return abs(y) < self.precision


@dataclass()
class StraightParamSpec(ParamSpec):
    right_limit: NUMBER
    left_limit: NUMBER

    def convert(self) -> tuple[Decimal, Decimal]:
        right, left = map(number_to_decimal, (self.right_limit, self.left_limit))
        return right, left


class StraightSolverABS(DifferentialSolver):
    def _solve(self, equation: AnyEquation, a: Decimal, f_a: Decimal, b: Decimal,
f_b: Decimal) -> Decimal:
        raise NotImplementedError()

    def solve(self, equation: AnyEquation, params: StraightParamSpec) -> Decimal:
        a, b = params.convert()
        f_a: Decimal = equation.function(a)
        f_b: Decimal = equation.function(b)
        if f_a == 0:
            return a
        if f_b == 0:
            return b
        if f_a * f_b > 0:
            raise ValueError("Can't find roots with given borders")
```

```
        step: int = 0
        xi = self._solve(equation, a, f_a, b, f_b)
        f_xi: Decimal = equation.function(xi)
        while not self.is_root(f_xi) and (self.max_steps is None or step <
self.max_steps):
            if f_a.is_signed() != f_xi.is_signed():
                b, f_b = xi, f_xi
            else:
                a, f_a = xi, f_xi
            xi = self._solve(equation, a, f_a, b, f_b)
            f_xi: Decimal = equation.function(xi)
            step += 1
        return xi


class BisectionSolver(StraightSolverABS):
    def _solve(self, equation: AnyEquation, a: Decimal, f_a: Decimal, b: Decimal,
f_b: Decimal) -> Decimal:
        return (a + b) / 2


class SecantSolver(StraightSolverABS):
    def _solve(self, equation: AnyEquation, a: Decimal, f_a: Decimal, b: Decimal,
f_b: Decimal) -> Decimal:
        return a - (f_a * (b - a)) / (f_b - f_a)


@dataclass()
class IterativeParamSpec(ParamSpec):
    initial_guess: NUMBER

    def convert(self) -> Decimal:
        return number_to_decimal(self.initial_guess)


class IterativeSolverABS(DifferentialSolver):
    def _solve(self, equation: AnyEquation, x_n: Decimal, f_x_n: Decimal) ->
Decimal:
        raise NotImplementedError()

    def solve(self, equation: AnyEquation, params: IterativeParamSpec) -> Decimal:
        step: int = 0
        x_n = params.convert()
        f_n = equation.function(x_n)
        while not self.is_root(f_n) and (self.max_steps is None or step <
self.max_steps):
            x_n = self._solve(equation, x_n, f_n)
            f_n = equation.function(x_n)
            step += 1
        return x_n


class NewtonSolver(IterativeSolverABS):
    def _solve(self, equation: AnyEquation, x_n: Decimal, f_n: Decimal) ->
```

```
Decimal:
        return x_n - f_n / equation.derivative(x_n)


class IterationSolver(IterativeSolverABS):
    def _solve(self, equation: AnyEquation, x_n: Decimal, f_n: Decimal) ->
Decimal:
        if equation.fixed_point is None:
            raise ValueError("")
        return equation.fixed_point(x_n)
```

Системы уравнений

```
class MultiEquation:
    def __init__(self, function: FunctionProtocol, derivative: DerivativeProtocol
= None, *, precision: int = 10):
        self.function: FunctionProtocol = function
        if derivative is not None:
            self.derivative: DerivativeProtocol = derivative
        self.precision: Decimal = Decimal(f"1E-{precision}")

    def derivative(self, x: Row, derive_by: int = 0) -> Decimal:
        x_moved: Row = x.copy()
        x_moved[derive_by] += self.precision
        return (self.function(x_moved) - self.function(x)) / self.precision


class EquationSystem:
    ...
    def solve(self, x: Row):
        step = 0
        delta_x: Row | None = None
        while (delta_x is None or abs(max(delta_x)) > self.precision) and step <
self.max_steps:
            jacobian = LinearEquationSystem([
                Row([equation.derivative(x, derive_by=i) for i in range(len(x))]
                    + [equation.function(x)])
                for equation in self])
            delta_x = jacobian.solve()[0]
            x -= delta_x
            step += 1
        return x
```

4. Примеры работы программы

Пример 1

$$x^2 + 4x - 5 = 0$$

```
BisectionSolver:               0.99999999998544808
SecantSolver:                  0.99999999998383437
NewtonSolver:                  1.0000000000011068
IterationSolver:               1.00000000001572864
```

Пример 2

$$e^x + 3x = 0$$

```
BisectionSolver:               -0.2576276530497367
SecantSolver:                  -0.2576276530497367
NewtonSolver:                  -0.2576276530497367
IterationSolver:               -0.2576276530497367
```

Пример 3 (система)

$$\begin{cases} 0.1x_1^2 + x_1 + 0.2x_2^2 - 0.3 = 0 \\ 0.2x_1^2 + x_2 - 0.1x_1x_2 - 0.7 = 0 \end{cases}$$

```
x1: 0.19641150552035911
x2: 0.70615418475557971
```

5. Вывод

Уравнения

В ходе решения лабораторной работы я изучил все четыре методы решения нелинейных уравнений из предложенных, их краткое сравнение:

- Метод половинного деления – один из самых неэффективных, но простых в реализации. От заданного отрезка мы переходим к отрезку половинной длинны, причём простой выбор середины – далеко не оптимальная стратегия.
- Метод хорд – усовершенствует метод половинного деления, учитывая при делении значения функции на краях отрезка, что даёт возможность быстрее уменьшать размер отрезка и обеспечивать более быструю работу
- Метод касательных – применим не всегда, но значительно увеличивает скорость схождения. Добавляется сложность реализации этому методу из-за необходимости вычислять производную функции.
- Метод простых итераций – по природе похож на метод касательных: более точные значения получаются на основе предыдущих, но теперь реализация осложняется не нахождением производной, а выведением функции $x = g(x)$.

Сложности всех вышеперечисленных методов методов линейно зависят от количества итераций.

Системы

Для решения систем уравнений было предложено два метода:

- Метод Ньютона, который обеспечивает большую скорость сходимости с ценой высокой сложности одной операции: $O(n^3)$ из-за того, что на каждой операции нужно решать систему линейных уравнений. Дополнительно осложняется необходимостью поиска частных производных для уравнений
- Метод итераций обеспечивает менее большую скорость сходимости, зато каждая операция даётся всего за $O(n^2)$. Осложняется, как и метод простых итераций для уравнений, выведением функции $x = g(x)$.