Helper.cs

```
namespace NonLinear.helper;

public struct Elem     /// Структура конечного элемента
{
    public double x1;  /// Координата начала КЭ
    public double x2;  /// Координата конца КЭ
    public double h;   /// Длина КЭ

    public Elem(double _x1, double _x2, double _h) {
        x1 = _x1; x2 = _x2; h = _h;
    }

    public override string ToString() => $"{x1,20} {x2,24} {h,24}";
}

public struct SLAU                 /// Структура СЛАУ
{
    public double[] di, dl, du;    /// Матрица
    public double[] q, q1;         /// Вектора решений
    public double[] f;             /// Правая часть
    public int MaxIter;            /// Максимальное количество итераций
    public double EPS;             /// Точность

    //* Обнуление СЛАУ (левой и правой частей)
    public void Clear() {
        Array.Clear(di, 0, di.Length);
        Array.Clear(dl, 0, dl.Length);
        Array.Clear(du, 0, du.Length);
        Array.Clear(f , 0, f.Length);
    }

    //* Перемножение матрицы на вектор
    public void Mult(double[] vec, out double[] res_vec) {
        res_vec = new double[vec.Length];
        res_vec[0] = di[0] * q[0] + du[0] * q[1];
        for (int i = 1; i < vec.Length - 1; i++) {
            res_vec[i]  = di[i] * q[i];
            res_vec[i] += du[i] * q[i + 1];
            res_vec[i] += dl[i] * q[i - 1];
        }
        res_vec[vec.Length - 1] = di[vec.Length - 1] * q[vec.Length - 1] + dl[vec.Length - 1] * q[vec.Length - 2];
    }

    //* Метод прогонки
    public double[] Progonka() {
        var alpha = new double[q.Length];
        var betta = new double[q.Length];

        // Подсчет альф и бетт
        alpha[0] = -du[0] / di[0];
        betta[0] = f[0] / di[0];
        for (int i = 1; i < q.Length; i++) {
            alpha[i] = -du[i] / (di[i] + dl[i] * alpha[i - 1]);
            betta[i] = (f[i] - dl[i] * betta[i - 1]) / (di[i] + dl[i] * alpha[i - 1]);
        }

        q[q.Length - 1] = betta[q.Length - 1];
        for (int i = q.Length - 2; i >= 0; i--)
            q[i] = alpha[i] * q[i + 1] + betta[i];
        return q;
    }
}

public static class Helper
```

```
{
    //: ***************** Перечисления ***************** :\\
    public enum Method {
        Iteration,
        Newton
    }
    //: ***************** Перечисления ***************** :\\

    //* Вычисление нормы вектора
    public static double Norm(double[] array) {
        double norm = 0;
        for (int i = 0; i < array.Count(); i++)
            norm += array[i] * array[i];
        return Sqrt(norm);
    }

    //* Окно помощи при запуске (если нет аргументов или по команде)
    public static void ShowHelp() {
        WriteLine("----Команды----                          \n" +
        "-help              - показать справку              \n" +
        "-i                 - входной файл                  \n" +
        "-m iteration       - метод простой итерации        \n" +
        "-m newton          - метод Ньютона                 \n" +
        "-func 'number'     - номер функции                 \n");
    }
}
```

Data.cs

```
namespace NonLinear;
public class Data
{
    //* Данные задачи
    public uint     countEl { get; set; }   /// Количество КЭ/отрезков
    public double   begin   { get; set; }   /// Начало отрезка
    public double   end     { get; set; }   /// Конец отрезка
    public double   k       { get; set; }   /// Коэффициент разрядки
    public double   gamma   { get; set; }   /// Коэффициенты гамма
    public double   betta   { get; set; }   /// Коэффициенты бетта
    public double[] kraev   { get; set; }   /// Краевые условия
    public double[] init_q  { get; set; }   /// Начальное приближение
}
```

Function.cs

```
namespace NonLinear;
public static class Function
{
    public static uint   NumberFunc;     /// Номер задачи

    //* Функция u(x,y)
    public static double Absolut(double x)
    {
        switch(NumberFunc)
        {
            case 1: /// test1-evenly (const)
            return 2.5;

            case 2: /// test2-polynom1
            return x;

            case 3: /// test3-polynom2
```

```
        return x*x;

        case 4: /// test4-polynom3
        return Pow(x, 3);

        case 5: /// test5_nopolynom
        return Sin(2*x);
    }
    return 0;
}

//* Функция f(x,y)
public static double Func(double x)
{
    switch(NumberFunc)
    {
        case 1: /// test1-evenly (const)
        return 5;

        case 2: /// test2-polynom1
        return 2*x;

        case 3: /// test3-polynom2
        return 2*x*x - 8*x - 8;

        case 4: /// test4-polynom3
        return -34*Pow(x, 3) - 24*x;

        case 5: /// test5_nopolynom
        return 16*Sin(2*x)*Cos(2*x);
    }
    return 0;
}

//* Лямбда от производной l(u`)
public static double Lambda(double diff, double x)
{
    switch(NumberFunc)
    {
        case 1: /// test1-evenly (const)
        return diff + 4;

        case 2: /// test2-polynom1
        return diff + 4;

        case 3: /// test3-polynom2
        return diff + 4;

        case 4: /// test4-polynom3
        return diff + 4;

        case 5: /// test5_nopolynom
        return diff + 4;
    }
    return 0;
}

//* Производная от Лямбды (в зависимости почему берем)
public static double DiffLambda(double diff, double x, double h,  uint num_q)
{
    switch(NumberFunc)
    {
        case 1: /// test1-evenly (const)
        return num_q switch
        {
            1 => -1/h,
            2 =>  1/h,
            _ => 0
```

```
            };

            case 2: /// test2-polynom1
            return num_q switch
            {
                1 => -1/h,
                2 =>  1/h,
                _ => 0
            };

            case 3: /// test3-polynom2
            return num_q switch
            {
                1 => -1/h,
                2 =>  1/h,
                _ => 0
            };

            case 4: /// test4-polynom3
            return num_q switch
            {
                1 => -1/h,
                2 =>  1/h,
                _ => 0
            };

            case 5: /// test5_nopolynom
            return num_q switch
            {
                1 => -1/h,
                2 =>  1/h,
                _ => 0
            };
        }
        return 0;
    }

    //* Производная абсолютной функции
    public static double Diff(double x)
    {
        switch(NumberFunc)
        {
            case 1: /// test1-evenly (const)
            return 0;

            case 2: /// test2-polynom1
            return 1;

            case 3: /// test3-polynom2
            return 2*x;

            case 4: /// test4-polynom3
            return 3*Pow(x, 2);

            case 5: /// test5_nopolynom
            return Cos(2*x);
        }
        return 0;
    }

}
```

PSolve.cs

```
public partial class Solve
{
    //* Генерация сетки
    private Elem[] Generate() {
        Elem[] elems = new Elem[data.countEl];

        // Подсчет начального шага
        double _h = data.k == 1
                    ? (data.end - data.begin) / data.countEl
                    : (1 - data.k) * (data.end - data.begin) / (1 - Pow(data.k, data.countEl));

        double x = data.begin; // Текущая позиция
        for (int i = 0; i < data.countEl && x <= data.end; i++) {
            elems[i] = new Elem(x, x + _h, _h);
            x  += _h;
            _h *= data.k;
        }

        StringBuilder grid = new StringBuilder();
        grid.Append($"{"x1",20} {"x2",24} {"h",24}\n");
        grid.Append(String.Join("\n", elems));
        File.WriteAllText($"{path}\\grid.txt", grid.ToString());
        return elems;
    }

    //* Путь к папке, где наход. решение задачи
    public void SetPath(string _path) {
        path = _path;
    }

    public partial void solve();
    private partial void memory();
}
```

Solve.cs

```
public partial class Solve
{
    protected Data    data;                    /// Данные задачи
    protected Method  method;                  /// Метод решения (Итерация или Ньютон)
    protected Elem[]  elems;                   /// Конечные элементы
    protected SLAU    slau;                    /// Структура СЛАУ
    protected string  path  { get; set; }      /// Путь к папке с задачей

    public Solve(Data _data, Method _method, uint _funcNumber) {
        Function.NumberFunc = _funcNumber;
        method              = _method    ;
        data                = _data      ;
    }

    public partial void solve() {
        elems = Generate();              //? Генерация сетки
        memory();                        //? Выделение памяти под матрица и вектора

        IMethods task = method switch
                    {
                        Method.Iteration => new Iteration(slau, data, elems),
                        Method.Newton    => new Newton(slau, data, elems),
                        _                => new Iteration(slau, data, elems)
                    };
        Directory.CreateDirectory($"{path}\\output");
```

```
        File.WriteAllText($"{path}\\output\\table_" + $"{task.GetName()}" + ".txt", task.solve());
    }

    private partial void memory() {
        slau.di   = new double[data.countEl + 1];
        slau.dl   = new double[data.countEl + 1];
        slau.du   = new double[data.countEl + 1];
        slau.q    = new double[data.countEl + 1];
        slau.q1   = new double[data.countEl + 1];
        slau.f    = new double[data.countEl + 1];
        // Записываем начальное приближение
        Array.Copy(data.init_q, slau.q, slau.q.Length);
    }
}
```

IMethods.cs

```
namespace NonLinear.methods;
public interface IMethods
{
    public string GetName();                                                    /// Возвращает имя метода
    public string solve();                                                      /// Главная функция решения
    public void global();                                                       /// Генерация глобальной матрицы
    public (double[][], double[]) local(int _i);                                /// Генерация локальной матрицы
    public void add_to_global(double[][] _mat, double[] _vec,  int _i);         /// Занесение локальной в глоабльную
    public void kraev();                                                        /// Учет краевых условий
    public void firstKraev(int _i);                                             /// Учет первого краевого
    public void secondKraev(int _i);                                            /// Учет второго краевого
    public void thirdKraev(int _i);                                             /// Учет третьего краевого
    public double[] newApprox();                                                /// Получение нового приближения (релаксация)
    public StringBuilder output();                                              /// Вывод решения
}
```

Iteration.cs

```
namespace NonLinear.methods;
public class Iteration : IMethods
{
    double w;                 /// Параметр релаксации
    private SLAU    slau;     /// структура СЛАУ
    private Data    data;     /// Данные задачи
    private Elem[] elems;     /// Конечные элементы

    public string GetName() => "Iteration";

    public Iteration(SLAU _slau, Data _data, Elem[] _elems, double _w = 1) {
        slau         = _slau;
        data         = _data;
        elems        = _elems;
        w            = _w;
        slau.MaxIter = 1000;
        slau.EPS     = 1e-10;
    }

    private bool Check(int iter, bool log = true) {
        global();
        slau.Mult(slau.q, out double[] vec_nev);
        vec_nev = slau.f.Zip(vec_nev, (f, s) => f - s).ToArray();
        double value = Norm(vec_nev) / Norm(slau.f);

        if (log)
```

```
        WriteLine($"{iter,5} : {value,10}");

    if (iter  > slau.MaxIter) return false;
    if (value < slau.EPS)     return false;

    return true;
}


//: ***** Реализация функций интерфейса ***** :\\
public string solve() {
    int Iter = 0; //? Количество итераций

    do {
        Array.Copy(slau.q, slau.q1, slau.q.Length);     //? q1 = q
        global();                                       //? Генерация глобальной матрицы
        slau.q = slau.Progonka();                       //? Решаем СЛАУ методом Прогонки
        slau.q = newApprox();                           //? Получаем новое приближение (релаксация)
    } while (Check(++Iter));

    return output().ToString();
}

public void global() {
    slau.Clear();                                               // Очищаем СЛАУ
    for (int index = 0; index < data.countEl; index++) {        // Проход по всем КЭ
        (double[][] mat, double[] vec) = local(index);          // Генерируем локальную матрицу и вектор
        add_to_global(mat, vec, index);                         // Заносим локальное в глобальное
    }
    kraev();                                                    // Учет краевых
}

public (double[][], double[]) local(int i) {
    var local_vec = new double[2];
    var local_mat = new double[2][];
    for (int j = 0; j < 2; j++) local_mat[j] = new double[2];

    double lam_arg = (slau.q[i + 1] - slau.q[i]) / elems[i].h;

    // + Матрица жесткости
    double value = (Lambda(lam_arg, elems[i].x1) + Lambda(lam_arg, elems[i].x2)) / (2*elems[i].h);
    for (int j = 0; j < 2; j++)
        for (int k = 0; k < 2; k++)
            local_mat[j][k] = j == k
                            ?  value
                            : -value;

    // + Матрица масс
    value = data.gamma * elems[i].h / 6.0;
    for (int j = 0; j < 2; j++)
        for (int k = 0; k < 2; k++)
            local_mat[j][k] += j == k
                            ?  2*value
                            :    value;

    // Правая часть
    var f = new double[2] { Func(elems[i].x1), Func(elems[i].x2) };
    for (int j = 0, k = 1; j < 2; j++, k--)
        local_vec[j] = (elems[i].h / 6.0) * (2*f[j] + f[k]);

    return (local_mat, local_vec);
}

public void add_to_global(double[][] mat, double[] vec, int i) {
    slau.di[i]     += mat[0][0];
    slau.di[i + 1] += mat[1][1];
    slau.du[i]     += mat[0][1];
    slau.dl[i + 1] += mat[1][0];
```

```
        slau.f[i]      += vec[0];
        slau.f[i + 1]  += vec[1];
    }

    public void kraev() {
        for (int i = 0; i < data.kraev.Length; i++) {
            switch (data.kraev[i])
            {
                case 1: firstKraev(i);  break;
                case 2: secondKraev(i); break;
                case 3: thirdKraev(i);  break;
            }
        }
    }

    public void firstKraev(int i) {
        switch(i) {
            // Левая граница
            case 0:
                slau.di[0] = 1;
                slau.du[0] = 0;
                slau.f[0]  = Absolut(elems[0].x1);
            break;

            // Правая граница
            case 1:
                slau.di[data.countEl] = 1;
                slau.dl[data.countEl] = 0;
                slau.f [data.countEl] = Absolut(elems[data.countEl - 1].x2);
            break;
        }
    }

    public void secondKraev(int i) {
        switch(i) {
            // Левая граница
            case 0:
                double lam_arg = (slau.q[1] - slau.q[0]) / elems[0].h;
                slau.f[0] -= Lambda(lam_arg, elems[0].x1) * Diff(elems[0].x1);
            break;

            // Правая граница
            case 1:
                lam_arg = (slau.q[data.countEl] - slau.q[data.countEl - 1]) / elems[data.countEl - 1].h;
                slau.f[data.countEl] += Lambda(lam_arg, elems[data.countEl - 1].x2) * Diff(elems[data.countEl - 1].x2);
            break;
        }
    }

    public void thirdKraev(int i)
    {
        switch(i) {
            // Левая граница
            case 0:
                slau.di[0] += data.betta;
                double lam_arg = (slau.q[1] - slau.q[0]) / elems[0].h;
                double res = Lambda(lam_arg, elems[0].x1) * (-1) * Diff(elems[0].x1);
                slau.f[0] += data.betta * (res + data.betta * Absolut(elems[0].x1));
            break;

            // Правая граница
            case 1:
                slau.di[data.countEl] += data.betta;
                lam_arg = (slau.q[data.countEl] - slau.q[data.countEl - 1]) / elems[data.countEl - 1].h;
                res = Lambda(lam_arg, elems[data.countEl - 1].x2) * Diff(elems[data.countEl - 1].x2);
                slau.f[data.countEl] += data.betta * (res + data.betta * Absolut(elems[data.countEl - 1].x2));
            break;
        }
```

```
    }

    public double[] newApprox() {
        return slau.q.Zip(slau.q1, (f, s) => w*f + (1 - w)*s).ToArray();
    }

    public StringBuilder output() {
        var table = new StringBuilder();

        string margin = String.Join("", Enumerable.Repeat("-", 16));

        table.Append(String.Join("", Enumerable.Repeat("-", 52)) + "\n");
        table.Append($"|U{" ",-14} | U`{" ",-12} | |U`- U| {" ",-7}|\n");
        table.Append($"|" + margin + "|" + margin + "|" + margin + "|\n");

        for (int i = 0; i < data.countEl; i++)
            table.Append($"|{String.Format("{0,16}", slau.q[i].ToString("E6"))}" +
                        $"|{String.Format("{0,16}", Absolut(elems[i].x1).ToString("E6"))}"   +
                        $"|{String.Format("{0,16}", Abs(Absolut(elems[i].x1) - slau.q[i]).ToString("E6"))}|\n");


        table.Append($"|{String.Format("{0,16}", slau.q[data.countEl].ToString("E6"))}" +
                    $"|{String.Format("{0,16}", Absolut(elems[data.countEl - 1].x2).ToString("E6"))}"   +
                    $"|{String.Format("{0,16}", Abs(Absolut(elems[data.countEl - 1].x2) - slau.q[data.countEl]).ToString("

        table.Append(String.Join("", Enumerable.Repeat("-", 52)) + "\n");

        return table;
    }
}
```

Newton.cs

```
namespace NonLinear.methods;
public class Newton : IMethods
{
    double w;                 /// Параметр релаксации
    private SLAU    slau;     /// структура СЛАУ
    private Data    data;     /// Данные задачи
    private Elem[] elems;     /// Конечные элементы

    public string GetName() => "Newton";

    public Newton(SLAU _slau, Data _data, Elem[] _elems, double _w = 1) {
        slau         = _slau;
        data         = _data;
        elems        = _elems;
        w            = _w;
        slau.MaxIter = 1000;
        slau.EPS     = 1e-10;
    }

    private bool Check(int iter, bool log = true) {
        global();
        slau.Mult(slau.q, out double[] vec_nev);
        vec_nev = slau.f.Zip(vec_nev, (f, s) => f - s).ToArray();
        double value = Norm(vec_nev) / Norm(slau.f);

        if (log)
            WriteLine($"{iter,5} : {value,10}");

        if (iter  > slau.MaxIter) return false;
        if (value < slau.EPS)     return false;

        return true;
```

```
}

void AddNewton(ref double[] mat_newton, int i) {
    double lam_arg   = (slau.q[i + 1] - slau.q[i]) / elems[i].h;
    double diff_l_q1_x1 = DiffLambda(lam_arg, elems[i].x1, elems[i].h, 1);
    double diff_l_q1_x2 = DiffLambda(lam_arg, elems[i].x2, elems[i].h, 1);
    double diff_l_q2_x1 = DiffLambda(lam_arg, elems[i].x1, elems[i].h, 2);
    double diff_l_q2_x2 = DiffLambda(lam_arg, elems[i].x2, elems[i].h, 2);

    // Матрица 1x4
    // A11_q1=A22_q1
    // A11_q2=A22_q2
    // A12_q1=A21_q1
    // A12_q2=A21_q2
    mat_newton[0] =  (diff_l_q1_x1 + diff_l_q1_x2) / (2 * elems[i].h);
    mat_newton[1] =  (diff_l_q2_x1 + diff_l_q2_x2) / (2 * elems[i].h);
    mat_newton[2] = -(diff_l_q1_x1 + diff_l_q1_x2) / (2 * elems[i].h);
    mat_newton[3] = -(diff_l_q2_x1 + diff_l_q2_x2) / (2 * elems[i].h);
}

//: ***** Реализация функций интерфейса ***** :\\
public string solve() {
    int Iter = 0; //? Количество итераций

    do {
        Array.Copy(slau.q, slau.q1, slau.q.Length);     //? q1 = q
        global();                                       //? Генерация глобальной матрицы
        slau.q = slau.Progonka();                       //? Решаем СЛАУ методом Прогонки
        slau.q = newApprox();                           //? Получаем новое приближение (релаксация)
    } while (Check(++Iter));

    return output().ToString();
}

public void global() {
    slau.Clear();                                               // Очищаем СЛАУ
    for (int index = 0; index < data.countEl; index++) {        // Проход по всем КЭ
        (double[][] mat, double[] vec) = local(index);          // Генерируем локальную матрицу и вектор
        add_to_global(mat, vec, index);                         // Заносим локальное в глобальное
    }
    kraev();                                                    // Учет краевых
}

public (double[][], double[]) local(int i) {
    var local_vec  = new double[2];
    var local_mat  = new double[2][];
    var mat_newton = new double[4];
    for (int j = 0; j < 2; j++) local_mat[j] = new double[2];

    double lam_arg = (slau.q[i + 1] - slau.q[i]) / elems[i].h;
    AddNewton(ref mat_newton, i);

    // + Матрица жесткости
    double value = (Lambda(lam_arg, elems[i].x1) + Lambda(lam_arg, elems[i].x2)) / (2*elems[i].h);
    local_mat[0][0] =  value + (mat_newton[0]*slau.q[i] + mat_newton[2]*slau.q[i + 1]);
    local_mat[0][1] = -value + (mat_newton[1]*slau.q[i] + mat_newton[3]*slau.q[i + 1]);
    local_mat[1][0] = -value + (mat_newton[2]*slau.q[i] + mat_newton[0]*slau.q[i + 1]);
    local_mat[1][1] =  value + (mat_newton[3]*slau.q[i] + mat_newton[1]*slau.q[i + 1]);

    // + Матрица масс
    value = (data.gamma * elems[i].h) / 6.0;
    for (int j = 0; j < 2; j++)
        for (int k = 0; k < 2; k++)
            local_mat[j][k] += j == k
                            ?  2*value
                            :    value;

    // Правая часть
```

```
        var f = new double[2] { Func(elems[i].x1), Func(elems[i].x2) };
        double add_newton_b0 = slau.q[i]     * (mat_newton[0]*slau.q[i] + mat_newton[1]*slau.q[i + 1]) +
                               slau.q[i + 1] * (mat_newton[2]*slau.q[i] + mat_newton[3]*slau.q[i + 1]);
        double add_newton_b1 = slau.q[i]     * (mat_newton[2]*slau.q[i] + mat_newton[3]*slau.q[i + 1]) +
                               slau.q[i + 1] * (mat_newton[0]*slau.q[i] + mat_newton[1]*slau.q[i + 1]);

        local_vec[0] = (elems[i].h / 6.0) * (2*f[0] + f[1]) + add_newton_b0;
        local_vec[1] = (elems[i].h / 6.0) * (2*f[1] + f[0]) + add_newton_b1;



        return (local_mat, local_vec);
    }

    public void add_to_global(double[][] mat, double[] vec, int i) {
        slau.di[i]     += mat[0][0];
        slau.di[i + 1] += mat[1][1];
        slau.du[i]     += mat[0][1];
        slau.dl[i + 1] += mat[1][0];
        slau.f[i]      += vec[0];
        slau.f[i + 1]  += vec[1];
    }

    public void kraev() {
        for (int i = 0; i < data.kraev.Length; i++) {
            switch (data.kraev[i])
            {
                case 1: firstKraev(i);  break;
                case 2: secondKraev(i); break;
                case 3: thirdKraev(i);  break;
            }
        }
    }

    public void firstKraev(int i) {
        switch(i) {
            // Левая граница
            case 0:
                slau.di[0] = 1;
                slau.du[0] = 0;
                slau.f[0]  = Absolut(elems[0].x1);
            break;

            // Правая граница
            case 1:
                slau.di[data.countEl] = 1;
                slau.dl[data.countEl] = 0;
                slau.f [data.countEl] = Absolut(elems[data.countEl - 1].x2);
            break;
        }
    }

    public void secondKraev(int i) {
        switch(i) {
            // Левая граница
            case 0:
                double lam_arg = (slau.q[1] - slau.q[0]) / elems[0].h;
                slau.f[0] -= Lambda(lam_arg, elems[0].x1) * Diff(elems[0].x1);
            break;

            // Правая граница
            case 1:
                lam_arg = (slau.q[data.countEl] - slau.q[data.countEl - 1]) / elems[data.countEl - 1].h;
                slau.f[data.countEl] += Lambda(lam_arg, elems[data.countEl - 1].x2) * Diff(elems[data.countEl - 1].x2);
            break;
        }
    }
```

```
    public void thirdKraev(int i)
    {
        switch(i) {
            // Левая граница
            case 0:
                slau.di[0] += data.betta;
                double lam_arg = (slau.q[1] - slau.q[0]) / elems[0].h;
                double res = Lambda(lam_arg, elems[0].x1) * (-1) * Diff(elems[0].x1);
                slau.f[0] += data.betta * (res + data.betta * Absolut(elems[0].x1));
            break;

            // Правая граница
            case 1:
                slau.di[data.countEl] += data.betta;
                lam_arg = (slau.q[data.countEl] - slau.q[data.countEl - 1]) / elems[data.countEl - 1].h;
                res = Lambda(lam_arg, elems[data.countEl - 1].x2) * Diff(elems[data.countEl - 1].x2);
                slau.f[data.countEl] += data.betta * (res + data.betta * Absolut(elems[data.countEl - 1].x2));
            break;
        }
    }

    public double[] newApprox() {
        return slau.q.Zip(slau.q1, (f, s) => w*f + (1 - w)*s).ToArray();
    }

    public StringBuilder output() {
        var table = new StringBuilder();

        string margin = String.Join("", Enumerable.Repeat("-", 16));

        table.Append(String.Join("", Enumerable.Repeat("-", 52)) + "\n");
        table.Append($"|U{" ",-14} | U`{" ",-12} | |U`- U| {" ",-7}|\n");
        table.Append($"|" + margin + "|" + margin + "|" + margin + "|\n");

        for (int i = 0; i < data.countEl; i++)
            table.Append($"|{String.Format("{0,16}", slau.q[i].ToString("E6"))}" +
                        $"|{String.Format("{0,16}", Absolut(elems[i].x1).ToString("E6"))}"   +
                        $"|{String.Format("{0,16}", Abs(Absolut(elems[i].x1) - slau.q[i]).ToString("E6"))}|\n");

        table.Append($"|{String.Format("{0,16}", slau.q[data.countEl].ToString("E6"))}" +
                    $"|{String.Format("{0,16}", Absolut(elems[data.countEl - 1].x2).ToString("E6"))}"   +
                    $"|{String.Format("{0,16}", Abs(Absolut(elems[data.countEl - 1].x2) - slau.q[data.countEl]).ToString("

        table.Append(String.Join("", Enumerable.Repeat("-", 52)) + "\n");

        return table;
    }
}
```

Program.cs

```
try {

    if (args.Length == 0) throw new ArgumentException("Not found arguments!");
    if (args[0] == "-help") {
        ShowHelp(); return;
    }

    string json = File.ReadAllText(args[1]);
    Data data = JsonConvert.DeserializeObject<Data>(json)!;
    if (data is null) throw new FileNotFoundException("File uncorrected!");

    Solve task = args[3] switch
    {
        "iteration" => new Solve(data, Method.Iteration, uint.Parse(args[5])),
```

```
        "newton"    => new Solve(data, Method.Newton,    uint.Parse(args[5])),
        _           => new Solve(data, Method.Iteration, uint.Parse(args[5]))
    };
    task.SetPath(Path.GetDirectoryName(args[1])!);
    task.solve();
}
catch (FileNotFoundException ex) {
    WriteLine(ex.Message);
}
catch (ArgumentException ex) {
    ShowHelp();
    WriteLine(ex.Message);
}
```