

Липецкий государственный технический университет

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА № 4

по дисциплине «Численные методы»

«Численные методы решения

нелинейных уравнений и систем уравнений»

Студент

Группа АС 21-1

подпись, дата

Станиславчук С.М.

Руководитель

Д.т.н, профессор кафедры АСУ

подпись, дата

Седых И.А.

Липецк 2023 г.

Содержание:

2. Задание кафедры.
3. Ход работы.
4. Выводы и сравнения результатов.
5. Полный код программ.

2. Задание кафедры

Вариант 10

1. Графически локализуите корень уравнения $f(x) = 0$ из таблицы 1 на начальном отрезке длиной не менее 1, обозначив его $[a, b]$. На отрезке $[a, b]$:

1.1 Найдите корень уравнения $f(x) = 0$ методом дихотомии с точностью $\epsilon = 1e-6$

1.2 Найдите корень уравнения $f(x) = 0$ методом Ньютона (касательных) с точностью $\epsilon = 1e-6$

1.3 Найдите корень уравнения $f(x) = 0$ методом простой итерации с точностью $\epsilon = 1e-6$, выбрав в качестве начального приближения x_0 один из концов начального отрезка (не 0!)

1.4 Найдите корень уравнения $f(x) = 0$ методом Стеффенсена с точностью $\epsilon = 1e-6$

2. Графически локализуите корни системы нелинейных уравнений $F_1(x_1, x_2) = 0$; $F_2(x_1, x_2) = 0$ из таблицы 3 на начальных отрезках по каждой переменной длиной не менее 1, обозначив их соответственно $[a_1, b_1]$ и $[a_2, b_2]$. На прямоугольнике $[[a_1, b_1]; [a_2, b_2]]$:

2.1 Найдите корень системы нелинейных уравнений методом Ньютона с точностью $\epsilon = 1e-6$

2.2 Найдите корень системы нелинейных уравнений двухступенчатым методом Ньютона с точностью $\epsilon = 1e-6$

2.3 Найдите корень системы нелинейных уравнений методом Зейделя с точностью $\epsilon = 1e-6$

2.4 Найдите корень системы нелинейных уравнений методом градиентного спуска с шагом $\alpha = 0.3$ (не слишком ли большой?) для нахождения минимума функции $\Phi(x_1, x_2) = F_1^2(x_1, x_2) + F_2^2(x_1, x_2)$ с точностью $\epsilon = 1e-6$

В заключении привести сравнительную таблицу результатов и сделать выводы.

10	$f(x) = 0,5 \exp(-\sqrt{x}) - 0,2\sqrt{x^3} + 2$
----	--------------------------------------------------

Уравнение для задания 1.

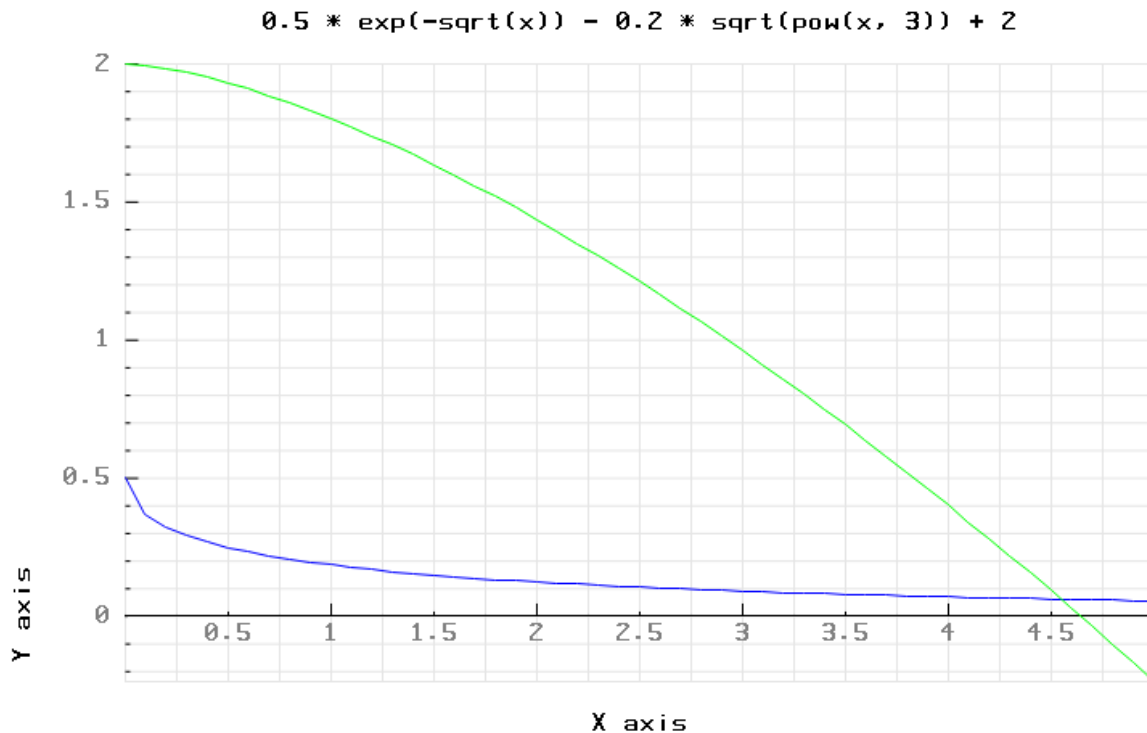
10	$\sin(0.5x_1 + x_2) - 1.2x_1 - 1 = 0$ $x_1^2 + x_2^2 - 1 = 0$
----	------------------------------------------------------------------

Система уравнений для задания 2

3. Ход работы

1) Графическая локализация

Выполнена при помощи C++ библиотеки pbPlots



Из рисунка видно, что точка пересечения приблизительно равна ~ 4.7 по оси X и ~ 0.1 по оси Y.

1.1) Метод дихотомии

Метод дихотомии — это численный метод для поиска корня унимодальной функции. Это означает, что функция имеет один единственный максимум или минимум на заданном интервале, и что она монотонно возрастает или убывает на этом интервале.

Принцип метода дихотомии заключается в том, что интервал, на котором ищется корень, последовательно делится на две равные части. Затем проверяется, на какой из половин интервала функция имеет знаки на концах разных знаков. Если это так, то корень находится на этой половине интервала, и процесс продолжается для этой половины. Если это не так, то процесс продолжается для другой половины интервала.

Алгоритм программы:

```
while (b - a >= eps) { // Все это делаем до тех пор, пока не достигнем
уровня погрешности
    double c = (a + b) / 2;
    if (f(a) * f(c) < 0) { // Проверяем "в какую сторону идти"
        b = c; " идем туда "
    }
    else {
```

```

        a = c; " или идем сюда "
    }
}
return (a + b) / 2;

```

Результат программы:

```

Number of dichotomy method ticks: 23
Dichotomy root = 4.729095399380

```

1.2) Метод Ньютона

Метод Ньютона, также известный как метод касательных, является численным методом для поиска корней дифференцируемой функции. Этот метод использует локальную информацию о функции в окрестности текущей точки для приближенного нахождения корня.

Основная идея метода Ньютона заключается в том, что мы начинаем с начального приближения для корня, и затем используем информацию о производной функции в этой точке, чтобы определить следующую точку, более близкую к корню. Метод состоит из последовательных итераций, на каждой из которых используется текущее приближение для нахождения следующего.

Формула для нахождения следующего приближения в методе Ньютона имеет вид:

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

Алгоритм:

```

double x_next = x - f(x) / df(x);
while (abs(x_next - x) >= eps) {
    x = x_next;
    x_next = x - f(x) / df(x);
}
return x_next;

```

Результат программы:

```

Number of Newton's method ticks: 4
Newton root = 4.729095647080

```

1.3) Метод простой итерации

Для решения уравнения $f(x) = 0$, метод простой итерации преобразует уравнение к виду $x = g(x)$, где $g(x)$ - некоторая функция. Затем метод начинает с некоторого начального значения x_0 и последовательно вычисляет

$$x_1 = g(x_0), x_2 = g(x_1), \dots, x_n = g(x_{n-1}).$$

Если последовательность x_n сходится к решению уравнения $f(x) = 0$, то x_n является приближенным решением уравнения с заданной точностью.

Для сходимости метода простой итерации необходимо, чтобы функция $g(x)$ была непрерывной и имела производную, а ее производная была меньше единицы по модулю на всей области сходимости.

Алгоритм:

```
double x1 = phi(x0);
while (fabs(x1 - x0) > eps) {
    x0 = x1;
    x1 = phi(x0);
}
return x1;
```

Результат программы:

```
Number of simple iteration method ticks: 2
Simple iteration root = 4.729095647080
```

1.4) Метод Стеффенсена

Метод Стеффенсена - это итерационный численный метод для приближенного решения уравнений вида $f(x) = 0$. Он основан на идее использования приближенного значения производной функции для улучшения скорости сходимости метода простой итерации.

Шаги в методе Стеффенсена:

1. Задать начальное приближение x_0 и погрешность ϵ .

2. На каждой итерации i вычислить следующее приближение x_i по формуле:

$$x_i = x_{i-1} - (f(x_{i-1}))^2 / (f(x_{i-1}) + f(x_{i-1}) - f(x_{i-1})))$$

3. Повторять шаг 2, пока $|x_i - x_{i-1}| > \epsilon$.

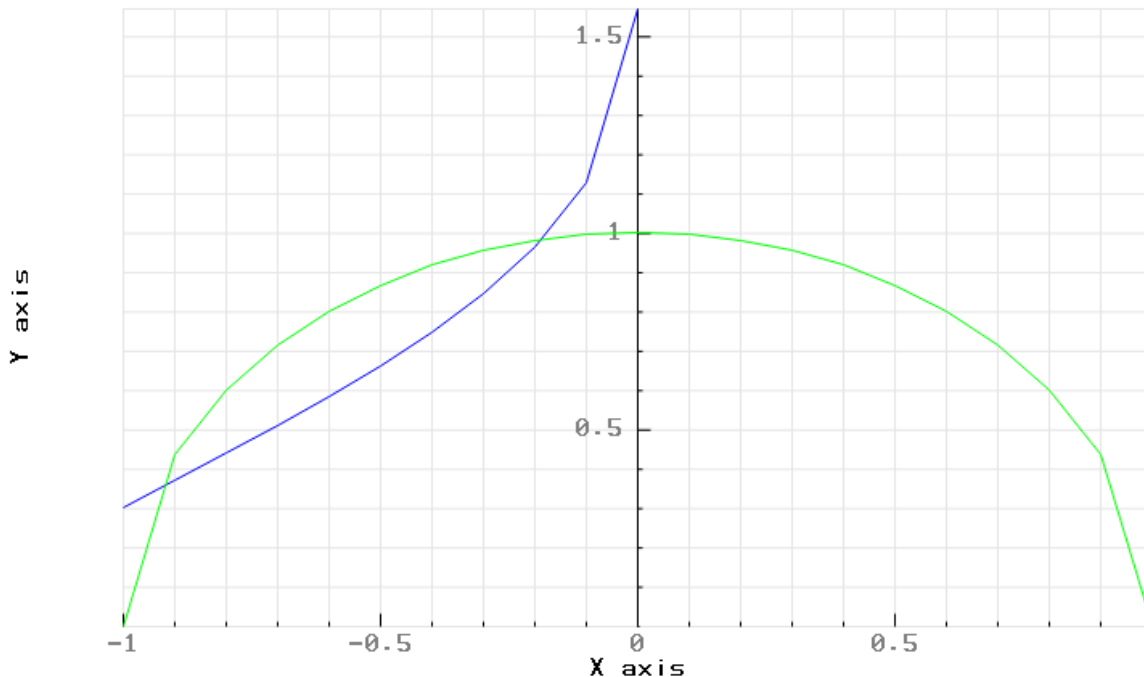
Алгоритм:

```
double x0, x1 = x, fpx;
do {
    x0 = x1;
    fpx = (f(x0 + f(x0)) - f(x0));
    x1 = x0 - (pow(f(x0), 2)) / fpx;
} while (abs(f(x1)) > eps);
return x0;
```

```
Number of ticks: 3
Steffensen's method root = 4.729362361082
```

2) Графическая локализация

`asin(1.2 * x1 + 1) - 0.5 * x1 (B) && sqrt(1 - pow(x1, 2)) (G)`



Из рисунка видно, что точка пересечения функций приблизительно равна ~ -0.2 по оси X и ~ 1 по оси Y.

2.1) Метод Ньютона

Метод Ньютона для систем нелинейных уравнений является численным методом нахождения приближенного решения системы уравнений $f(x) = 0$, где $f(x)$ - вектор-функция, определенная на n -мерном пространстве.

Шаги метода Ньютона для системы нелинейных уравнений:

Выбрать начальное приближение x_0 .

1. Вычислить матрицу Якоби $J(x_0)$ и вектор $f(x_0)$.

2. Решить систему линейных уравнений $J(x_0) * \text{deltax} = -f(x_0)$ для нахождения приращения deltax .

3. Вычислить следующее приближение $x_1 = x_0 + \text{deltax}$.

4. Проверить, достигнута ли требуемая точность. Если нет, вернуться к шагу 2, используя x_1 как новое начальное приближение.

Алгоритм:

```
do {  
    _f1 = f1(x1, x2);  
    _f2 = f2(x1, x2);  
  
    J[0][0] = df1_dx1(x1, x2); // Считаем якобиан  
    J[0][1] = df1_dx2(x1, x2);
```

```

J[1][0] = df2_dx1(x1);
J[1][1] = df2_dx2(x2);

detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

invJ[0][0] = J[1][1] / detJ; // Считаем обратный якобиан
invJ[0][1] = -J[0][1] / detJ;
invJ[1][0] = -J[1][0] / detJ;
invJ[1][1] = J[0][0] / detJ;

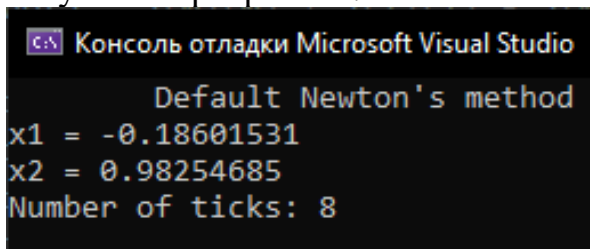
deltaX1 = invJ[0][0] * (-_f1) + invJ[0][1] * (-_f2);
deltaX2 = invJ[1][0] * (-_f1) + invJ[1][1] * (-_f2);

x1 += deltaX1;
x2 += deltaX2;

norm = fabs(deltaX1 + deltaX2);
} while (norm > eps);

```

Результат программы:



```

C:\> Консоль отладки Microsoft Visual Studio

Default Newton's method
x1 = -0.18601531
x2 = 0.98254685
Number of ticks: 8

```

2.2) Двухступенчатый метод Ньютона

Двухступенчатый метод Ньютона - это метод численного решения систем нелинейных уравнений, который позволяет уменьшить количество операций, связанных с вычислением обратной матрицы Якоби на каждой итерации метода Ньютона.

Обычный метод Ньютона для систем нелинейных уравнений на каждой итерации требует вычисления и обращения матрицы Якоби, что может быть затратным по времени и ресурсам, особенно для больших систем. В двухступенчатом методе Ньютона матрица Якоби вычисляется только через определенное количество итераций, а до этого на первых нескольких итерациях используется приближенная матрица, например, единичная матрица. Это может сократить количество операций и сделать метод более эффективным.

Алгоритм:

```

do {
    double f1_val = f1(x1, x2);
    double f2_val = f2(x1, x2);
    J[0][0] = df1_dx1(x1, x2);
    J[0][1] = df1_dx2(x1, x2);
    J[1][0] = df2_dx1(x1);
    J[1][1] = df2_dx2(x2);
    double detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

    dx1 = (-f1_val * J[1][1] + f2_val * J[0][1]) / detJ;
    dx2 = (f1_val * J[1][0] - f2_val * J[0][0]) / detJ;

    x1 += dx1;
    x2 += dx2;
}

```



```

}
while (fabs(dx1) > eps && fabs(dx2) > eps);

```

```

Two-step newton method
x1 = -0.18601531
x2 = 0.98254685
Number of ticks: 7

```

2.3) Метод Зейделя

Метод Зейделя является итерационным методом для решения систем нелинейных уравнений. Он похож на метод простых итераций, но на каждой итерации он использует уже вычисленные значения, чтобы улучшить приближение к решению.

Для решения системы нелинейных уравнений методом Зейделя на каждой итерации мы используем следующую формулу:

$$\begin{aligned}
 x1_{i+1} &= f1(x1_i, x2_i, \dots, xn_i) \\
 x2_{i+1} &= f2(x1_{i+1}, x2_i, \dots, xn_i) \\
 &\dots \\
 xn_{i+1} &= fn(x1_{i+1}, x2_{i+1}, \dots, xn_i)
 \end{aligned}$$

где $x1_i, x2_i, \dots, xn_i$ - текущее приближение к решению на i -ой итерации, $x1_{i+1}, x2_{i+1}, \dots, xn_{i+1}$ - новое приближение на $i+1$ -ой итерации, $f1, f2, \dots, fn$ - функции, описывающие систему уравнений.

Алгоритм:

```

do {
    x1_prev = x1;
    x2_prev = x2;

    // Вычисляем новое значение x1
    x1 = (sin(0.5 * x1_prev + x2) - 1) / 1.2;

    // Вычисляем новое значение x2
    x2 = sqrt(1 - pow(x1, 2));

    _f1 = sin(0.5 * x1 + x2) - 1.2 * x1 - 1;
    _f2 = pow(x1, 2) + pow(x2, 2) - 1;

    iter++;
} while (_f1 > eps || _f2 > eps);

```

Результат программы:

```

Seidel method
x1 = -0.18601590
x2 = 0.98254673
Number if ticks: 16

```

2.4) Метод градиентного спуска

Метод градиентного спуска - это итерационный алгоритм оптимизации, который используется для нахождения минимума (или максимума) некоторой функции. Он основан на использовании градиента (вектора частных производных) функции в качестве направления наискорейшего убывания.

Алгоритм работает следующим образом:

1. Задаётся начальное приближение x_0 и шаг обучения α .
2. Вычисляется градиент функции $f(x)$ в точке x_k : $\text{grad}(f(x_k))$.
3. Вычисляется следующее приближение x_{k+1} как x_k минус α умножить на градиент функции $f(x_k)$: $x_{k+1} = x_k - \alpha * \text{grad}(f(x_k))$.
4. Повторяем шаги 2-3 до тех пор, пока значение функции $f(x)$ не будет достаточно мало или пока не будет достигнуто максимальное число итераций.

Алгоритм:

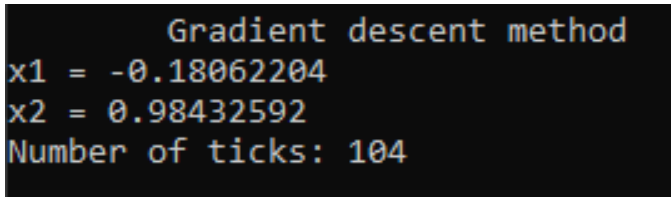
```
while (i < max_iterations) {
    dx1 = dPhi_dx1(x1, x2, h);
    dx2 = dPhi_dx2(x1, x2, h);

    temp_x1 = x1 - step_size * dx1;
    temp_x2 = x2 - step_size * dx2;

    if (fabs(Phi(temp_x1, temp_x2) - Phi(x1, x2)) < eps) {
        break;
    }

    x1 = temp_x1;
    x2 = temp_x2;

    i++;
}
```



```
Gradient descent method
x1 = -0.18062204
x2 = 0.98432592
Number of ticks: 104
```

5.

Таблица и выводы для задания 1:

	Дихотомия	Ньютон	Прост итерация	Стеффенсен
x	4.7290954	4.7290956	4.7290956	4.7293624
n итераций	23	4	2	3

Вывод:

Метод простой итерации сходится быстрее всех, потребовав 2 итерации.

Метод дихотомии требует большего числа итераций (23), чтобы достичь решения, чем остальные методы.

Таким образом, можно рекомендовать использовать метод простой итерации для этой задачи, так как он обеспечивает наиболее быструю сходимость к

решению. Однако, для этого метода требуется дополнительная подготовка, например, подсчет значения функции Φ .

Таблица и выводы для задания 2:

	Ньютон	2х_Ньютон	Зейдель	Град. спуск
x1	-0.18601531	-0.18601531	-0.18601590	-0.18062204
x2	0.98254685	0.98254685	0.98254673	0.98432592
n _{итераций}	8	7	16	104

Метод Ньютона и двухступенчатый метод Ньютона требуют меньшего числа итераций для достижения заданной точности по сравнению с методами Зейделя и градиентного спуска.

Для данной системы уравнений наиболее эффективным методом является метод двухступенчатого Ньютона, так как он сходится к решению быстрее, чем метод Ньютона, и требует меньшего числа итераций, чем метод Зейделя и градиентный спуск, однако метод требует расчета производных.

Программы:

Задание 1.

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <pbPlots.cpp>
#include <supportLib.cpp>
using namespace std;

double f(double x) {
    return 0.5 * exp(-sqrt(x)) - 0.2 * sqrt(pow(x, 3)) + 2;
}
double df(double x) {
    return (-0.25 * exp(-sqrt(x))) / sqrt(x) - 0.3 * sqrt(x);
}
double phi(double x) {
    return -2/(0.5 * exp(-sqrt(x)) - 0.2 * sqrt(pow(x,3)));
}
double phi_test(double x) {
    return x - ((1.0/df(x))*(f(x)));
}

// Для графического представления функций разделим нашу f на f1 и f2
double f1(double x) {
    return 0.5 * exp(-sqrt(x));
}
double f2(double x) {
    return - 0.2 * sqrt(pow(x,3)) + 2;
}

double dichotomy(double a, double b, double eps) {
    int ticks = 0;
    while (b - a >= eps) {
        double c = (a + b) / 2;
        if (f(a) * f(c) < 0) {
            b = c;
        }
        else {
            a = c;
        }
        ticks++;
    }
    cout << "Number of dichotomy method ticks: " << ticks;
    return (a + b) / 2;
}

double newton(double x, double eps) {
    double x_next = x - f(x) / df(x);
    int ticks = 0;
    while (abs(x_next - x) >= eps) {
        x = x_next;
        x_next = x - f(x) / df(x);
        ticks++;
    }
    cout << "\nNumber of Newton's method ticks: " << ticks;
    return x_next;
}

double simple_iteration(double x0, double eps) {
    double x1 = phi_test(x0);
    int ticks = 0;
    while (fabs(x1 - x0) > eps) {
        x0 = x1;
        x1 = phi_test(x0);
    }
}
```

```

        ticks++;
    }
    cout << "\nNumber of simple iteration method ticks: " << ticks;
    return x1;
}

double steffensen(double x, double eps) {
    double x0, x1 = x, fpx;
    int ticks = 0;
    do {
        x0 = x1;
        fpx = (f(x0 + f(x0)) - f(x0));
        x1 = x0 - (pow(f(x0),2)) / fpx;

        ticks++;
    } while (abs(f(x1)) > eps);
    cout << "\nNumber of ticks: " << ticks;
    return x0;
}

int main() {
    cout << setprecision(7) << fixed;
    double eps = 1e-6;
    double a = 0;
    double b = 5;

    double dihRoot = dichotomy(a, b, eps);
    cout << "\nDichotomy root = " << dihRoot << "\n";

    double newtRoot = newton((b-a)/2, eps);
    cout << "\nNewton root = " << newtRoot << "\n";

    double simpleIterRoot = simple_iteration(b, eps); // where 1st argument is
x0 cout << "\nSimple iteration root = " << simpleIterRoot << "\n";

    double steffensenRoot = steffensen((b-a)/2, eps); // where 1st argument is
x0 cout << "\nSteffensen's method root = " << steffensenRoot << "\n";

#pragma region Plot

    vector<double> f1_array;
    vector<double> f2_array;
    vector<double> x_array;
    double temp = a;
    while (temp <= b) {
        x_array.push_back(temp);
        f1_array.push_back(f1(temp));
        f2_array.push_back(f2(temp));
        temp += 0.1;
    }

    RGBABitmapImageReference* imageReference = CreateRGBABitmapImageReference();

    ScatterPlotSeries* series1 = GetDefaultScatterPlotSeriesSettings();
    series1->xs = &x_array;
    series1->ys = &f1_array;
    series1->lineType = toVector(L"solid");
    series1->color = CreateRGBColor(0, 0, 1);

    ScatterPlotSeries* series2 = GetDefaultScatterPlotSeriesSettings();
    series2->xs = &x_array;
    series2->ys = &f2_array;
    series2->lineType = toVector(L"solid");

```

```

series2->color = CreateRGBColor(0, 1, 0);

ScatterPlotSettings* settings = GetDefaultScatterPlotSettings();
settings->width = 800;
settings->height = 480;
settings->title = toVector(L"0.5 * exp(-sqrt(x)) - 0.2 * sqrt(pow(x, 3)) +
2");
settings->xLabel = toVector(L"X axis");
settings->yLabel = toVector(L"Y axis");
settings->scatterPlotSeries->push_back(series1);
settings->scatterPlotSeries->push_back(series2);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>* pngData = ConvertToPNG(imageReference->image);
WriteToFile(pngData, "plot.png");
cout << "\nPlot generated\n";
DeleteImage(imageReference->image);

#pragma endregion

return 0;
}

```

Программа 2:

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <pbPlots.cpp>
#include <supportLib.cpp>

#define eps 1e-6

using namespace std;

void displayMatr2D(double matr[2][2]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << matr[i][j] << "\t";
        }
        cout << "\n";
    }
}

#pragma region Functions
double f1(double x1, double x2) {
    return sin(0.5 * x1 + x2) - 1.2 * x1 - 1;
}
double f2(double x1, double x2) {
    return pow(x1, 2) + pow(x2, 2) - 1;
}
double FplotF1(double x1) {
    return asin(1.2 * x1 + 1) - 0.5 * x1;
}
double FplotF2(double x1) {
    return sqrt(1-pow(x1, 2));
}
#pragma endregion

#pragma region For_Newtons
double df1_dx1(double x1, double x2) {
    return 0.5 * cos(0.5 * x1 + x2) - 1.2;
}
double df1_dx2(double x1, double x2) {
    return cos(0.5 * x1 + x2);
}
double df2_dx1(double x1) {
    return 2 * x1;
}
double df2_dx2(double x2) {
    return 2 * x2;
}
#pragma endregion

/* Example
double f1(double x1, double x2) {
    return sin(x1 + 1.5) - x2 + 2.9;
}
double f2(double x1, double x2) {
    return cos(x2 - 2) + x1;
}
double df1_dx1(double x1, double x2) {
    return cos(x1 + 1.5);
}
double df1_dx2(double x1, double x2) {
    return -1;
}
double df2_dx1(double x1) {
    return 1;
}
```

```

}
double df2_dx2(double x2) {
    return -sin(x2 - 2);
}

// For Seidel method
double g1(double x1, double x2) {
    return -cos(x2-2);
}
double g2(double x1, double x2) {
    return sin(x1+1.5)+2.9;
}
*/

#pragma region Default_Newton

void jacobian(double x1, double x2, double& j11, double& j12, double& j21,
double& j22)
{
    j11 = df1_dx1(x1, x2);
    j12 = df1_dx2(x1, x2);
    j21 = df2_dx1(x1);
    j22 = df2_dx2(x2);
}
void inverse(double j11, double j12, double j21, double j22, double& inv_j11,
double& inv_j12, double& inv_j21, double& inv_j22)
{
    double det = j11 * j22 - j12 * j21;

    inv_j11 = j22 / det;
    inv_j12 = -j12 / det;
    inv_j21 = -j21 / det;
    inv_j22 = j11 / det;
}

void newton_default(double x1, double x2)
{
    double _f1, _f2, J[2][2], detJ, invJ[2][2], deltaX1, deltaX2, norm;
    int tick = 0;

    do {
        _f1 = f1(x1, x2);
        _f2 = f2(x1, x2);

        J[0][0] = df1_dx1(x1, x2);
        J[0][1] = df1_dx2(x1, x2);
        J[1][0] = df2_dx1(x1);
        J[1][1] = df2_dx2(x2);

        detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

        invJ[0][0] = J[1][1] / detJ;
        invJ[0][1] = -J[0][1] / detJ;
        invJ[1][0] = -J[1][0] / detJ;
        invJ[1][1] = J[0][0] / detJ;

        deltaX1 = invJ[0][0] * (-_f1) + invJ[0][1] * (-_f2);
        deltaX2 = invJ[1][0] * (-_f1) + invJ[1][1] * (-_f2);

        x1 += deltaX1;
        x2 += deltaX2;

        norm = fabs(deltaX1 + deltaX2);

        tick++;
    } while (norm > 1e-6);
}

```



```

    } while (norm > eps);

    cout << "\tDefault Newton's method";
    //cout << "\nJacobian matrix: \n";
    //displayMatr2D(J);
    //cout << "\nInversed Jacobean matrix: \n";
    //displayMatr2D(invJ);
    cout << "\nx1 = " << x1 << "\nx2 = " << x2 << "\n";
    cout << "Number of ticks: " << tick << "\n";
}

#pragma endregion

#pragma region Two_step_newton
void Two_step_newton(double x1, double x2)
{
    int tick = 0;
    double dx1, dx2;
    double J[2][2];
    do {
        double f1_val = f1(x1, x2);
        double f2_val = f2(x1, x2);
        J[0][0] = df1_dx1(x1, x2);
        J[0][1] = df1_dx2(x1, x2);
        J[1][0] = df2_dx1(x1, x2);
        J[1][1] = df2_dx2(x1, x2);
        double detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

        dx1 = (-f1_val * J[1][1] + f2_val * J[0][1]) / detJ;
        dx2 = (f1_val * J[1][0] - f2_val * J[0][0]) / detJ;

        x1 += dx1;
        x2 += dx2;

        tick++;
    }
    while (fabs(dx1) > eps && fabs(dx2) > eps);

    cout << "\tTwo-step newton method\n";
    cout << "x1 = " << x1 << "\n";
    cout << "x2 = " << x2 << "\n";
    cout << "Number of ticks: " << tick << "\n";
}
#pragma endregion

#pragma region Seidel
// For Seidel method
double g1(double x1, double x2) {
    return (sin(0.5 * x1 + x2) - 1) / (1.2); // pow(sin(1.2 * x1 + 1), -1) - 0.5
    * x1 // 2 * (asin(1.2 * x1 + 1) - x2) // (sin(0.5 * x1 + x2) - 1) / 1.2
}
double g2(double x1, double x2) {
    return sqrt(1 - pow(x1, 2)); // sqrt(1 - pow(x1, 2)) // sqrt(1 - pow(x2, 2))
}

void seidel_method(double x1, double x2)
{
    double x1_prev, x2_prev;
    double diff;
    double _f1, _f2;
    int iter = 0;
    do {
        x1_prev = x1;
        x2_prev = x2;

        // Вычисляем новое значение x1

```

```

    x1 = (sin(0.5 * x1_prev + x2) - 1) / 1.2;

    // Вычисляем новое значение x2
    x2 = sqrt(1 - pow(x1, 2));

    _f1 = sin(0.5 * x1 + x2) - 1.2 * x1 - 1;
    _f2 = pow(x1, 2) + pow(x2, 2) - 1;

    iter++;
} while (_f1 > eps || _f2 > eps);

cout << "\tSeidel method \n";
cout << "x1 = " << x1 << endl;
cout << "x2 = " << x2 << endl;
cout << "Number of ticks: " << iter << endl;
}

#pragma endregion

#pragma region Gradient_Descent

// Функция для вычисления значения  $\Phi(x1, x2)$ 
double Phi(double x1, double x2) {
    return pow(f1(x1, x2), 2) + pow(f2(x1, x2), 2);
}

// Функции для вычисления производных  $\Phi(x1, x2)$  по  $x1$  и  $x2$ 
double dPhi_dx1(double x1, double x2, double h) {
    return (Phi(x1 + h, x2) - Phi(x1, x2)) / h;
}

double dPhi_dx2(double x1, double x2, double h) {
    return (Phi(x1, x2 + h) - Phi(x1, x2)) / h;
}

// Функция градиентного спуска
void gradient_descent(double x1, double x2, double step_size) {
    int i = 0, max_iterations = 10000;
    double dx1, dx2, temp_x1, temp_x2;
    double h = 0.0001; // шаг для вычисления производных

    while (i < max_iterations) {
        dx1 = dPhi_dx1(x1, x2, h);
        dx2 = dPhi_dx2(x1, x2, h);

        temp_x1 = x1 - step_size * dx1;
        temp_x2 = x2 - step_size * dx2;

        if (fabs(Phi(temp_x1, temp_x2) - Phi(x1, x2)) < eps) {
            break;
        }

        x1 = temp_x1;
        x2 = temp_x2;

        i++;
    }

    cout << "\tGradient descent method\n";
    cout << "x1 = " << x1 << "\n";
    cout << "x2 = " << x2 << "\n";
    cout << "Number of ticks: " << i << "\n";
}

#pragma endregion

```

```

int main() {
    double x1 = 2.0;
    double x2 = 2.0;
    cout << setprecision(8) << fixed;
    newton_default(x1, x2);
    cout << "\n\n";
    Two_step_newton(x1, x2);
    cout << "\n\n";
    seidel_method(x1, x2);
    cout << "\n\n";
    gradient_descent(0.0, 1.0, 0.03);
    cout << "\n\n";

#pragma region Plot
    double a = -1;
    double b = 1;
    vector<double> f1_array;
    vector<double> f2_array;
    vector<double> x_array;
    double temp = a;
    while (temp <= b) {
        x_array.push_back(temp);
        f1_array.push_back(FplotF1(temp));
        f2_array.push_back(FplotF2(temp));
        temp += 0.1;
    }

    RGBABitmapImageReference* imageReference = CreateRGBABitmapImageReference();

    ScatterPlotSeries* series1 = GetDefaultScatterPlotSeriesSettings();
    series1->xs = &x_array;
    series1->ys = &f1_array;
    series1->lineType = toVector(L"solid");
    series1->color = CreateRGBColor(0, 0, 1);

    ScatterPlotSeries* series2 = GetDefaultScatterPlotSeriesSettings();
    series2->xs = &x_array;
    series2->ys = &f2_array;
    series2->lineType = toVector(L"solid");
    series2->color = CreateRGBColor(0, 1, 0);

    ScatterPlotSettings* settings = GetDefaultScatterPlotSettings();
    settings->width = 800;
    settings->height = 480;
    settings->title = toVector(L"asin(1.2 * x1 + 1) - 0.5 * x1 (B) && sqrt(1 -
pow(x1, 2)) (G)");
    settings->xLabel = toVector(L"X axis");
    settings->yLabel = toVector(L"Y axis");
    settings->scatterPlotSeries->push_back(series1);
    settings->scatterPlotSeries->push_back(series2);

    DrawScatterPlotFromSettings(imageReference, settings, NULL);

    vector<double>* pngData = ConvertToPNG(imageReference->image);
    WriteToFile(pngData, "plot.png");
    cout << "\nPlot generated\n";
    DeleteImage(imageReference->image);

#pragma endregion

    return 0;
}

```