**Міністерство освіти і науки України**

**Чернівецький національний університет імені Юрія Федьковича**

**Факультет математики та інформатики**

**Кафедра математичного моделювання**

**Звіт**

**до лабораторної роботи №1**

**з «Комп'ютерного моделювання жорстких процесів та систем»**

**Виконали:**

**cтуденти 301-Б групи**

**Мар’янчук О.О. ,**

**Колісник Г.Б.**

**Викладач:**

**кандидат фіз.-мат. наук, асистент**

**Іліка С.А.**

**Чернівці 2023**

**Лабораторна робота № 1**

**Тема:** Багатокрокові різницеві схеми

**Завдання:**

Використовуючи задану багатокрокову р.с. знайти розв’язок модельної задачі Коші для системи диференціальних рівнянь. Вхід у таблицю підрахувати за однокроковою р.с. У випадку неявної р.с. розв’язок уточнювати методом простої ітерації або методом Ньютона. Друк результатів здійснити в точках кратних 0.1.

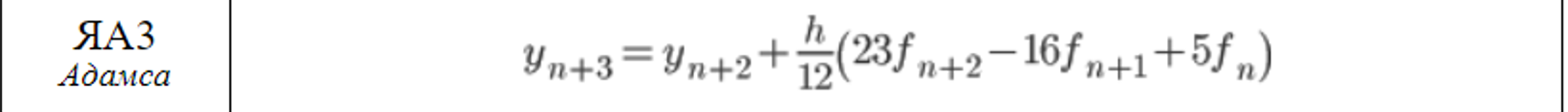
**Варіант 6**

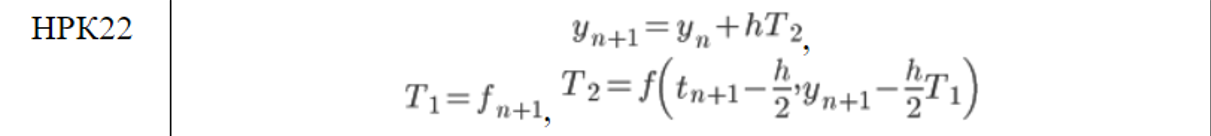
****

**2 тип**

ЯА3+ НРК22+ Проста ітерація і Ньютона

**Різницеві схеми:**

****



**Код до завдання:**

#include <iostream>

#include <iomanip>

#include <vector>

#include <cassert>

#include <cmath>

#include <functional>

using namespace std;

struct Vec2t {

    double t;

    double x;

    double y;

    Vec2t() :

        x(0.0),

        y(0.0),

        t(0.0)

    {};

    Vec2t(double \_t, double \_x, double \_y) :

        t(\_t), x(\_x), y(\_y)

    {};

    friend Vec2t operator+ (const Vec2t& v1, const Vec2t& v2)

    {

        return Vec2t(

            v1.t + v2.t,

            v1.x + v2.x,

            v1.y + v2.y

        );

    }

    friend Vec2t operator- (const Vec2t& v1, const Vec2t& v2)

    {

        return Vec2t(

            v1.t - v2.t,

            v1.x - v2.x,

            v1.y - v2.y

        );

    }

    friend Vec2t operator\* (double c, const Vec2t& v)

    {

        return Vec2t(

            c \* v.t,

            c \* v.x,

            c \* v.y

        );

    }

};

struct Vec2 {

    double x;

    double y;

    Vec2() :

        x(0.0), y(0.0)

    {};

    Vec2(double \_x, double \_y) :

        x(\_x), y(\_y)

    {};

    Vec2(const Vec2t& v) :

        x(v.x), y(v.y)

    {};

    friend Vec2 operator+ (const Vec2& v1, const Vec2& v2)

    {

        return Vec2(

            v1.x + v2.x,

            v1.y + v2.y

        );

    }

    friend Vec2 operator- (const Vec2& v1, const Vec2& v2)

    {

        return Vec2(

            v1.x - v2.x,

            v1.y - v2.y

        );

    }

    friend Vec2 operator\* (double c, const Vec2& v)

    {

        return Vec2(

            c \* v.x,

            c \* v.y

        );

    }

};

struct Mat2x2 {

    union {

        struct {

            double \_11;

            double \_12;

            double \_21;

            double \_22;

        };

        struct {

            double fx;

            double fy;

            double gx;

            double gy;

        };

    };

    Mat2x2(double e) :

        \_11(e),

        \_12(e),

        \_21(e),

        \_22(e)

    {};

    Mat2x2(double e11, double e12, double e21, double e22) :

        \_11(e11),

        \_12(e12),

        \_21(e21),

        \_22(e22)

    {};

    friend Vec2 operator\*(const Mat2x2& mat, const Vec2& vect)

    {

        return Vec2(mat.\_11 \* vect.x + mat.\_12 \* vect.y, mat.\_21 \* vect.x + mat.\_22 \* vect.y);

    };

    friend Mat2x2 operator\*(double c, const Mat2x2& mat)

    {

        return Mat2x2(c \* mat.\_11, c \* mat.\_12, c \* mat.\_21, c \* mat.\_22);

    }

    double determinant() {

        return \_11 \* \_22 - \_12 \* \_21;

    }

    Mat2x2 inverse() {

        return (1 / determinant()) \* Mat2x2(\_22, -\_12, -\_21, \_11);

    }

};

using TabFunc = std::vector<Vec2t>;

using Func = std::function<Vec2(const Vec2t&)>;

using FuncS = std::function<Vec2(double)>;

using AMat2x2 = std::function<Mat2x2(const Vec2t&)>;

class Solver

{

public:

    enum Status {

        NewRootFinded,

        MaxIterationReached,

        SufficientToleranceReached

    };

    Solver(uint32\_t \_maxIteration, double \_tolerance, const Vec2t& \_lastRoot, const Func& \_system) :

        maxIteration(\_maxIteration),

        tolerance(\_tolerance),

        lastRoot(\_lastRoot),

        system(\_system)

    {};

    virtual Status nextRoot() = 0;

    Vec2t getLastRoot() const

    {

        return lastRoot;

    }

    uint32\_t getElapsedIteration() const {

        return iteration;

    }

    double getLastTolerance() const {

        return lastEps;

    }

protected:

    uint32\_t maxIteration;

    uint32\_t iteration = 0;

    double tolerance;

    Func system;

    Vec2t lastRoot;

    double lastEps = 0.0;

};

class SimpleIterationSolver : public Solver

{

public:

    SimpleIterationSolver(uint32\_t \_maxIteration, double \_tolerance, const Vec2t& \_lastRoot, const Func& \_system) :

        Solver(\_maxIteration, \_tolerance, \_lastRoot, \_system)

    {};

    Status nextRoot() override {

        if (iteration > maxIteration)

            return Status::MaxIterationReached;

        iteration++;

        Vec2 newRoot = system(lastRoot);

        lastEps = std::abs(newRoot.x - lastRoot.x) + std::abs(newRoot.y - lastRoot.y);

        lastRoot.x = newRoot.x;

        lastRoot.y = newRoot.y;

        if (lastEps < tolerance)

            return Status::SufficientToleranceReached;

        return Status::NewRootFinded;

    }

};

class NewtonIterationSolver : public Solver

{

public:

    NewtonIterationSolver(uint32\_t \_maxIteration, double \_tolerance, const Vec2t& \_lastRoot, const Func& \_system, const AMat2x2& \_jacobian) :

        Solver(\_maxIteration, \_tolerance, \_lastRoot, \_system), jacobian(\_jacobian)

    {};

    Status nextRoot() override {

        if (iteration > maxIteration)

            return Status::MaxIterationReached;

        iteration++;

        Vec2 newRoot;

        Mat2x2 test = jacobian(lastRoot).inverse();

        newRoot = lastRoot - jacobian(lastRoot).inverse() \* system(lastRoot);

        double curTolerance = std::abs(newRoot.x - lastRoot.x) + std::abs(newRoot.y - lastRoot.y);

        lastRoot.x = newRoot.x;

        lastRoot.y = newRoot.y;

        if (curTolerance < tolerance)

            return Status::SufficientToleranceReached;

        return Status::NewRootFinded;

    }

private:

    AMat2x2 jacobian;

};

TabFunc getTabFunc(const FuncS& func, double t0, double t1, double h) {

    TabFunc tab;

    uint32\_t count = (t1 - t0) / h;

    for (size\_t i = 0; i <= count; i++)

    {

        double t = t0 + i \* h;

        Vec2 valFunc = func(t);

        tab.push\_back(Vec2t(

            t,

            valFunc.x,

            valFunc.y

        ));

    }

    return tab;

}

void printCompTabFunc(const TabFunc& func1, const TabFunc& func2, uint32\_t width = 20, uint32\_t precision = 1, uint32\_t t\_precision = 2, uint32\_t diff\_precision = 6) {

    assert(func1.size() == func2.size());

    std::cout << '|' << std::setfill('-') << std::setw(width) << 't'

        << '|' << std::setw(width) << 'x'

        << '|' << std::setw(width) << 'y'

        << '|' << std::setw(width) << "standart x"

        << '|' << std::setw(width) << "standart y"

        << '|' << std::setw(width) << "diff x"

        << '|' << std::setw(width) << "diff y" << "|\n";

    std::cout << std::setfill(' ');

    for (size\_t i = 0; i < func1.size(); i++)

    {

        std::cout << '|' << std::setw(width) << std::fixed << std::setprecision(t\_precision) << func1[i].t;

        std::cout << '|' << std::setw(width) << std::fixed << std::setprecision(precision) << func1[i].x

            << '|' << std::setw(width) << std::fixed << std::setprecision(precision) << func1[i].y;

        std::cout << '|' << std::setw(width) << std::fixed << std::setprecision(precision) << func2[i].x

            << '|' << std::setw(width) << std::fixed << std::setprecision(precision) << func2[i].y;

        std::cout << '|' << std::setw(width) << std::fixed << std::setprecision(diff\_precision) << std::abs(func2[i].x - func1[i].x)

            << '|' << std::setw(width) << std::fixed << std::setprecision(diff\_precision) << std::abs(func2[i].y - func1[i].y) << "|\n";;

    }

}

class DifferenceScheme {

public:

    DifferenceScheme(double \_t0, double \_t1, const Vec2& \_v, double \_h, uint32\_t \_k, double tolerance, const Func& \_system, const AMat2x2& \_jacobian) :

        t0(\_t0),

        t1(\_t1),

        initPoint(\_v),

        h(\_h),

        k(\_k),

        eps(tolerance),

        system(\_system),

        jacobian(getSufficientJacobian(\_jacobian))

    {};

    TabFunc getRungeKutta22(uint32\_t countOfNodes = 3) {

        Vec2t p(t0, initPoint.x, initPoint.y);

        TabFunc t;

        t.push\_back(p);

        setOptimalH(p);

        t.push\_back(p);

        uint32\_t count = (t1 - t0) / h;

        for (uint32\_t i = 2; i <= count && i < countOfNodes; i++)

        {

            p.t = t0 + i \* h;

            SimpleIterationSolver solver(k, eps, p, getRungeKuttaScheme(p));

            while (solver.nextRoot() == Solver::Status::NewRootFinded);

            p = solver.getLastRoot();

            t.push\_back(p);

        }

        return t;

    }

    TabFunc getRungeKutta22Newton(uint32\_t countOfNodes = 3) {

        Vec2t p(t0, initPoint.x, initPoint.y);

        TabFunc t;

        t.push\_back(p);

        setOptimalNewtonH(p);

        t.push\_back(p);

        uint32\_t count = (t1 - t0) / h;

        for (uint32\_t i = 2; i <= count && i < countOfNodes; i++)

        {

            p.t = t0 + i \* h;

            NewtonIterationSolver solver(k, eps, p, getRungeKuttaSchemeNewton(p), jacobian);

            while (solver.nextRoot() == Solver::Status::NewRootFinded);

            p = solver.getLastRoot();

            t.push\_back(p);

        }

        return t;

    }

    TabFunc getAdams3(const TabFunc& \_tb) {

        assert(\_tb.size() == 3);

        TabFunc tb = \_tb;

        uint32\_t count = (t1 - t0) / h;

        for (uint32\_t i = 3; i <= count; i++)

        {

            uint32\_t l = tb.size() - 1;

            Vec2 n2(tb[l] + h / 12 \* (23 \* system(tb[l]) - 16 \* system(tb[l - 1]) + 5 \* system(tb[l - 2])));

            Vec2t n3(t0 + i \* h, n2.x, n2.y);

            tb.push\_back(n3);

        }

        return tb;

    }

    Func getRungeKuttaScheme(const Vec2t& v0) {

        return [this, &v0](const Vec2t& v) {

            Vec2 K1 = v - (h / 2) \* system(v);

            Vec2 K2 = system(Vec2t(v.t - h / 2., K1.x, K1.y));

            return v0 + h \* K2;

        };

    }

    Func getRungeKuttaSchemeNewton(const Vec2t& v0) {

        return [this, &v0](const Vec2t& v) {

            Vec2 K1 = v - (h / 2) \* system(v);

            Vec2 K2 = system(Vec2t(v.t - h / 2., K1.x, K1.y));

            return v - v0 - h \* K2;

        };

    }

    void setCurH(double \_h) {

        h = \_h;

    }

    double getCurH() const {

        return h;

    }

    void setOptimalH(Vec2t& root) {

        Vec2t lastRoot;

        const double MinH = 1e-3;

        {

            root.t = h;

            SimpleIterationSolver solver(k, eps, root, getRungeKuttaScheme(root));

            while (solver.nextRoot() == Solver::Status::NewRootFinded);

            lastRoot = solver.getLastRoot();

        }

        do

        {

            Vec2t сurRoot = root;

            h /= 2;

            сurRoot.t = h;

            SimpleIterationSolver solver1(k, eps, сurRoot, getRungeKuttaScheme(сurRoot));

            Solver::Status status1;

            while ((status1 = solver1.nextRoot()) == Solver::Status::NewRootFinded);

            сurRoot = solver1.getLastRoot();

            сurRoot.t += h;

            SimpleIterationSolver solver2(k, eps, сurRoot, getRungeKuttaScheme(сurRoot));

            Solver::Status status2;

            while ((status2 = solver2.nextRoot()) == Solver::Status::NewRootFinded);

            сurRoot = solver2.getLastRoot();

            double tolerance = getTolerance(сurRoot, lastRoot);

            lastRoot = solver1.getLastRoot();

            if (status1 == Solver::Status::SufficientToleranceReached &&

                status2 == Solver::Status::SufficientToleranceReached &&

                tolerance < eps)

                break;

        } while (h > MinH);

        root = lastRoot;

    }

    void setOptimalNewtonH(Vec2t& root) {

        Vec2t lastRoot;

        const double MinH = 1e-3;

        {

            root.t = h;

            NewtonIterationSolver solver(k, eps, root, getRungeKuttaSchemeNewton(root), jacobian);

            while (solver.nextRoot() == Solver::Status::NewRootFinded);

            lastRoot = solver.getLastRoot();

        }

        do

        {

            Vec2t сurRoot = root;

            h /= 2;

            сurRoot.t = h;

            NewtonIterationSolver solver1(k, eps, сurRoot, getRungeKuttaSchemeNewton(сurRoot), jacobian);

            Solver::Status status1;

            while ((status1 = solver1.nextRoot()) == Solver::Status::NewRootFinded);

            сurRoot = solver1.getLastRoot();

            сurRoot.t += h;

            NewtonIterationSolver solver2(k, eps, сurRoot, getRungeKuttaSchemeNewton(сurRoot), jacobian);

            Solver::Status status2;

            while ((status2 = solver2.nextRoot()) == Solver::Status::NewRootFinded);

            сurRoot = solver2.getLastRoot();

            double tolerance = getTolerance(сurRoot, lastRoot);

            lastRoot = solver1.getLastRoot();

            if (status1 == Solver::Status::SufficientToleranceReached &&

                status2 == Solver::Status::SufficientToleranceReached &&

                tolerance < eps)

                break;

        } while (h > MinH);

        root = lastRoot;

    }

    double getTolerance(const Vec2t& p1, const Vec2t& p2) const {

        return std::abs(p1.x - p2.x) + std::abs(p1.y - p2.y);

    }

    AMat2x2 getSufficientJacobian(const AMat2x2& jacobian) {

        return [this, jacobian](const Vec2t& v) -> Mat2x2 {

            Mat2x2 j = jacobian(v);

            Vec2 \_vw = v - (h / 2) \* system(v);

            Vec2t vw(v.t, \_vw.x, \_vw.y);

            Mat2x2 jvw = jacobian(vw);

            return Mat2x2(

                1 - h \* jvw.fx \* (1 - (h / 2) \* j.fx) + h \* jvw.fy \* ((h / 2) \* j.gx),

                h \* jvw.fx \* ((h / 2) \* j.fy) - h \* jvw.fy \* (1 - (h / 2) \* j.gy),

                -h \* jvw.gx \* (1 - (h / 2) \* j.fx) + h \* jvw.gy \* ((h / 2) \* j.gx),

                1 + h \* jvw.gx \* ((h / 2) \* j.fy) - h \* jvw.gy \* (1 - (h / 2) \* j.gy)

            );

        };

    }

private:

    double k;

    double t0;

    double t1;

    Vec2 initPoint;

    double h;

    double eps;

    Func system;

    AMat2x2 jacobian;

};

int main()

{

    DifferenceScheme ds(0.0, 1.0, Vec2(1.f, 1.f), 1, 100, 1e-3,

        [](const Vec2t& v) -> Vec2 {

            return Vec2(

                v.t / v.y,

                - v.t / v.x

            );

        },

        [](const Vec2t& v) -> Mat2x2 {

            return Mat2x2(

                0.0,

                - v.t / (v.y \* v.y),

                v.t / (v.x \* v.x),

                0.0

            );

        }

    );

    FuncS sampleSystem = [](double t) {

        constexpr double E = 2.71828182845904523536;

        return Vec2(

            std::pow(E, t \* t / 2),

            std::pow(E, -t \* t / 2)

        );

    };

    TabFunc tbSimple = ds.getRungeKutta22();

    tbSimple = ds.getAdams3(tbSimple);

    std::cout << "Simple Iteration : \n";

    std::cout << "Optimal H : " << ds.getCurH() << '\n';

    printCompTabFunc(tbSimple, getTabFunc(sampleSystem, 0.0, 1.0, ds.getCurH()));

    ds.setCurH(1);

    TabFunc tbNewton = ds.getRungeKutta22Newton();

    tbNewton = ds.getAdams3(tbNewton);

    std::cout << "Newton Method : \n";

    std::cout << "Optimal H : " << ds.getCurH() << '\n';

    printCompTabFunc(tbNewton, getTabFunc(sampleSystem, 0.0, 1.0, ds.getCurH()));

    return 0;

}

**Скріни виконання завдання:**

