```cpp
//Utility.h

#include <cmath>
#include <fstream>
#include <iostream>
#include <string>
#include <windows.h>
#include <vector>

class Save
{
public:
    //public constructor/s
    Save(const double h, const double k, const size_t M, const size_t N, const std::vector<std::vector<double>>& u, const std::string save_file_name);
};

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

class MidPoint
{
public:
    //public constructor/s
    MidPoint(const double h, const size_t M, double& midpoint);
};

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

class IniVector
{
public:
    //public constructor/s
    IniVector(const size_t M, std::vector<double>& vec);
};

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

class IniMatrix
{
public:
    //public constructor/s
    IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>& mat);
};

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

class InitialConditions
{
public:
    //public constructor/s
    InitialConditions(const double h, const size_t M, const size_t N, std::vector<std::vector<double>>& u, const double midpoint);
};
```

```cpp
//Utility.cpp

#include "0.0.Utility.h"

Save::Save(const double h, const double k, const size_t M, const size_t N, const std::vector<std::vector<double>>& u, const std::string save_file_name)
{
    std::ofstream out;
    out.open(save_file_name);
    if (out.is_open())
    {
        for (size_t i = 0; i < M; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                out << i * h << "," << j * k << "," << u[i][j] << "\n";
            }
        }
    }
    else
    {
        throw std::string("Could not save to file!");
    }
    out.close();
}

////////////////////////////////////////////////////////////////////////////////////////////////////

MidPoint::MidPoint(const double h, const size_t M, double& midpoint)
{
    midpoint = ((M - 1) * h) / 2;;
}

////////////////////////////////////////////////////////////////////////////////////////////////////

IniVector::IniVector(const size_t M, std::vector<double>& vec)
{
    vec.resize(M);
    for (size_t i = 0; i < M; i++)
    {
        vec[i] = 0;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////

IniMatrix::IniMatrix(const size_t M, const size_t N, std::vector<std::vector<double>>& mat)
{
    mat.resize(M, std::vector<double>(N));
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            mat[i][j] = 0;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////

InitialConditions::InitialConditions(const double h, const size_t M, const size_t N, std::vector<std::vector<double>>& vec, const double midpoint)
{
    for (size_t i = 0; i < M; i++)
```

```
63          {
64              vec[i][0] = exp(-pow((i * h - midpoint), 2));
65          }
66          for (size_t j = 0; j < N; j++)
67          {
68              vec[0][j] = 0;
69              vec[M - 1][j] = 0;
70          }
71      }
```

```cpp
//MatrixOperations.h

class MatrixCopy
{
public:
    MatrixCopy(const size_t M, const std::vector<std::vector<double>>& mat_from, std::
    vector<std::vector<double>>& mat_to);
};

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

class VectorCopy
{
public:
    VectorCopy(const size_t M, const std::vector<double>& vec_from, std::vector<double>&
    vec_to);
};
```

```cpp
//MatrixOperations.cpp

#include "0.0.Utility.h"
#include "0.1.MatrixOperations.h"

MatrixCopy::MatrixCopy(const size_t M, const std::vector<std::vector<double>>& mat_from, std::vector<std::vector<double>>& mat_to)
{
    for (size_t i = 0; i < M; i++)
    {
        for (size_t j = 0; j < M; j++)
        {
            mat_to[i][j] = mat_from[i][j];
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

VectorCopy::VectorCopy(const size_t M, const std::vector<double>& vec_from, std::vector<double>& vec_to)
{
    for (size_t i = 0; i < M; i++)
    {
        vec_to[i] = vec_from[i];
    }
}
```

```cpp
//LUGaussSpecial.h

class LUGaussSpecial
{
public:
    //public constructor/s
    LUGaussSpecial(const size_t M, const double s_0, const double s_1, const double s_2,
    std::vector<double>& b, std::vector<double>& x);
private:
    //private function/s
    void SpecialGaussianElimination(std::vector<double>& b, std::vector<double>& x);
    void ExtractTriVectors(const double s_0, const double s_1, const double s_2);
    //private variable/s
    const size_t M;
    std::vector<double> a{};
    std::vector<double> c{};
    std::vector<double> d{};
};
```

```cpp
//LUGaussSpecial.cpp

#include "0.0.Utility.h"
#include "0.1.MatrixOperations.h"
#include "0.2.LUGaussSpecial.h"

//public contstructor/s
LUGaussSpecial::LUGaussSpecial(const size_t M, const double s_0, const double s_1, const double s_2, std::vector<double>& b, std::vector<double>& x)
    : M(M)
{
    IniVector(M, a);
    IniVector(M, c);
    IniVector(M, d);

    ExtractTriVectors(s_0, s_1, s_2);
    SpecialGaussianElimination(b, x);
}
//private function/s
void LUGaussSpecial::SpecialGaussianElimination(std::vector<double>& b, std::vector<double>& x)
{
    for (size_t i = 1; i < M; i++)
    {
        d[i] = d[i] - (a[i - 1] / d[i - 1]) * c[i - 1];
        b[i] = b[i] - (a[i - 1] / d[i - 1]) * b[i - 1];
    }
    x[M - 1] = b[M - 1] / d[M - 1];
    for (int i = M - 2; i >= 0; i--)
    {
        x[i] = (b[i] - (c[i] * x[i + 1])) / d[i];
    }
}
void LUGaussSpecial::ExtractTriVectors(const double s_0, const double s_1, const double s_2)
{
    for (size_t i = 0; i < M; i++)
    {
        a[i] = s_0;
        d[i] = s_1;
        c[i] = s_2;
    }
}
```

```cpp
//Tri.h

class Tri
{
public:
    //public constructor/s
    Tri(double s_0, double s_1, double s_2, size_t M, size_t column, std::vector<double>&
     b, std::vector<std::vector<double>>& mat);
private:
    //private function/s
    void ExecuteTri(const double s_0, const double s_1, const double s_2, const size_t
    column, std::vector<double>& b, std::vector<std::vector<double>>& mat);
    void AssignResults(const size_t column, std::vector<std::vector<double>>& mat);
    //private variable/s
    const size_t M;
    std::vector<double> x{};
};
```

```cpp
//Tri.cpp

#include "0.0.Utility.h"
#include "0.2.LUGaussSpecial.h"
#include "0.3.Tri.h"

//public contstructor/s
Tri::Tri(double s_0, double s_1, double s_2, size_t M, size_t column, std::vector<double>& b, std::vector<std::vector<double>>& mat)
    : M(M)
{
    IniVector(M, x);

    ExecuteTri(s_0, s_1, s_2, column, b, mat);
}
//private function/s
void Tri::ExecuteTri(const double s_0, const double s_1, const double s_2, const size_t column, std::vector<double>& b, std::vector<std::vector<double>>& mat)
{
    LUGaussSpecial solve(M, s_0, s_1, s_2, b, x);
    AssignResults(column, mat);
}
void Tri::AssignResults(const size_t column, std::vector<std::vector<double>>& mat)
{
    for (size_t i = 1; i < M - 1; i++)
    {
        mat[i][column + 1] = x[i];
    }
}
```

```cpp
//Eq1.h

#pragma once

//par_u/par_t + par_u/par_x = 0  differential equation using Euler + Runge-Kutta 2nd order methods

class ParDiffEq1Euler
{
public:
    //public constructor/s
    ParDiffEq1Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};

    const double h;
    const double k;
};

class ParDiffEq1RK2
{
public:
    //public constructor/s
    ParDiffEq1RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    const double h;
    const double k;
};
```

```cpp
//Eq1.cpp

#include "0.0.Utility.h"
#include "1.Eq1.h"

//public contstructor/s
ParDiffEq1Euler::ParDiffEq1Euler(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq1Euler::Euler()
{
    const double s{ k / (2 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (u[i + 1][j] - u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq1/Euler_Eq_1_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq1RK2::ParDiffEq1RK2(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq1RK2::RK2()
{
    const double r{ k / (4 * h) };
    const double s{ k / (2 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r * (u[i + 1][j] - u[i - 1][j]);
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (v[i + 1][j + 1] - v[i - 1][j + 1]);
        }
    }

    std::string save_file_name{ "Results/Eq1/RK2_Eq_1_h_" + std::to_string(h) + "_k_" +
    std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}
```

```cpp
//Eq2.h

//par_u/par_t + u*par_u/par_x = 0 differential equation using Euler + Runge-Kutta 2nd order methods

class ParDiffEq2Euler
{
public:
    //public constructor/s
    ParDiffEq2Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};

    const double h;
    const double k;
};

class ParDiffEq2RK2
{
public:
    //public constructor/s
    ParDiffEq2RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 1001 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    const double h;
    const double k;
};
```

```cpp
//Eq2.cpp

#include "0.0.Utility.h"
#include "2.Eq2.h"

//public contstructor/s
ParDiffEq2Euler::ParDiffEq2Euler(double h, double k)
    : h(h), k(k)
{
    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq2Euler::Euler()
{
    const double s{ k / (4 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
        }
    }

    std::string save_file_name{ "Results/Eq2/Euler_Eq_2_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq2RK2::ParDiffEq2RK2(double h, double k)
    : h(h), k(k)
{
    MidPoint ini_midpoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions ini_initial_conditions_u(h, M, N, u, midpoint);
    InitialConditions ini_initial_conditions_v(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq2RK2::RK2()
{
    const double r{ k / (8 * h) };
    const double s{ k / (4 * h) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2));
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s * (pow(v[i + 1][j + 1], 2) - pow(v[i - 1][j + 1], 2
            ));
        }
    }

    std::string save_file_name{ "Results/Eq2/RK2_Eq_2_h_" + std::to_string(h) + "_k_" +
    std::to_string(k) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}
```

```cpp
//Eq3.h

//par_u/par_t - beta*par_u^2/par_x^2 = 0 differential equation using Euler + Runge-Kutta 2nd order + implicit methods

class ParDiffEq3Euler
{
public:
    //public constructor/s
    ParDiffEq3Euler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEq3RK2
{
public:
    //public constructor/s
    ParDiffEq3RK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEq3Implicit
{
public:
    //public constructor/s
    ParDiffEq3Implicit(double h, double k);
    //public function/s
    void Implicit();
private:
    //private function/s
    void SetVectorb(const size_t column);
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<double> b{};

    double beta;
    const double h;
    const double k;
};
```

```cpp
//Eq3.cpp

#include "0.0.Utility.h"
#include "0.3.Tri.h"
#include "3.Eq3.h"

//public contstructor/s
ParDiffEq3Euler::ParDiffEq3Euler(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEq3Euler::Euler()
{
    const double s{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] + s * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq3/Euler_Eq_3_h_" + std::to_string(h) + "_k_" +
     std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEq3RK2::ParDiffEq3RK2(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEq3RK2::RK2()
{
    const double r{ (beta * k) / (2 * pow(h,2)) };
    const double s{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] + r * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] + s * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j +
             1]);
        }
    }

```

```
67      std::string save_file_name{ "Results/Eq3/RK2_Eq_3_h_" + std::to_string(h) + "_k_" +
        std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
68
69      Save(h, k, M, N, u, save_file_name);
70  }
71
72  ////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////
73
74  //public contstructor/s
75  ParDiffEq3Implicit::ParDiffEq3Implicit(double h, double k)
76      : h(h), k(k)
77  {
78      beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
79
80      MidPoint(h, M, midpoint);
81      IniMatrix(M, M, u);
82      IniVector(M, b);
83      InitialConditions(h, M, M, u, midpoint);
84  }
85  //public function/s
86  void ParDiffEq3Implicit::Implicit()
87  {
88      const double s{ (beta * k) / pow(h,2) };
89      const double s_0{ -s };
90      const double s_1{ 1 + 2 * s };
91      const double s_2{ -s };;
92
93      for (size_t j = 0; j < M-1; j++)
94      {
95          SetVectorb(j);
96          Tri solve(s_0, s_1, s_2, M, j, b, u);
97      }
98
99      std::string save_file_name{ "Results/Eq3/Implicit_Eq_3_h_" + std::to_string(h) +
        "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
100
101     Save(h, k, M, M, u, save_file_name);
102 }
103 //private function/s
104 void ParDiffEq3Implicit::SetVectorb(const size_t column)
105 {
106     for (size_t i = 0; i < M; i++)
107     {
108         b[i] = u[i][column];
109     }
110 }
```

```cpp
//Burgers.h

//Burgers' equation: par_u/par_t + u*par_u/par_x - beta*par_u^2/par_x^2=0 = 0 using Euler + Runge-Kutta 2nd order + implicit methods

class ParDiffEqBurgersEuler
{
public:
    //public constructor/s
    ParDiffEqBurgersEuler(double h, double k);
    //public function/s
    void Euler();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};

    double beta;
    double h;
    double k;
};

class ParDiffEqBurgersRK2
{
public:
    //public constructor/s
    ParDiffEqBurgersRK2(double h, double k);
    //public function/s
    void RK2();
private:
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    const size_t N{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<std::vector<double>> v{};

    double beta;
    const double h;
    const double k;
};

class ParDiffEqBurgersImplicit
{
public:
    //public constructor/s
    ParDiffEqBurgersImplicit(double h, double k);
    //public function/s
    void Implicit();
private:
    //private function/s
    void SetVectorb(const size_t column);
    //private variable/s
    double midpoint{};
    const size_t M{ 101 };
    std::vector<std::vector<double>> u{};
    std::vector<double> b{};

    double beta;
    const double h;
    const double k;
};
```

```cpp
//Burgers.cpp

#include "0.0.Utility.h"
#include "0.3.Tri.h"
#include "4.Burgers.h"

//public contstructor/s
ParDiffEqBurgersEuler::ParDiffEqBurgersEuler(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    InitialConditions ini_inital_conditions_u(h, M, N, u, midpoint);
}
//public function/s
void ParDiffEqBurgersEuler::Euler()
{
    const double s_0{ k / (4 * h) };
    const double s_1{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            u[i][j + 1] = u[i][j] - s_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
            s_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
    }

    std::string save_file_name{ "Results/Eq4/Euler_Burgers_h_" + std::to_string(h) +
    "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };

    Save(h, k, M, N, u, save_file_name);
}

////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////

//public contstructor/s
ParDiffEqBurgersRK2::ParDiffEqBurgersRK2(double h, double k)
    : h(h), k(k)
{
    beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/

    MidPoint(h, M, midpoint);
    IniMatrix(M, N, u);
    IniMatrix(M, N, v);
    InitialConditions(h, M, N, u, midpoint);
    InitialConditions(h, M, N, v, midpoint);
}
//public function/s
void ParDiffEqBurgersRK2::RK2()
{
    const double r_0{ k / (8 * h) };
    const double r_1{ (beta * k) / (2 * pow(h,2)) };

    const double s_0{ k / (4 * h) };
    const double s_1{ (beta * k) / pow(h,2) };

    for (size_t j = 0; j < N - 1; j++)
    {
        for (size_t i = 1; i < M - 1; i++)
        {
            v[i][j + 1] = u[i][j] - r_0 * (pow(u[i + 1][j], 2) - pow(u[i - 1][j], 2)) +
            r_1 * (u[i + 1][j] - 2 * u[i][j] + u[i - 1][j]);
        }
        for (size_t i = 1; i < M - 1; i++)
```

```
66          {
67              u[i][j + 1] = u[i][j] - s_0 * (pow(v[i + 1][j + 1], 2) - pow(v[i - 1][j + 1],
                2)) + s_1 * (v[i + 1][j + 1] - 2 * v[i][j + 1] + v[i - 1][j + 1]);
68          }
69      }
70
71      std::string save_file_name{ "Results/Eq4/RK2_Burgers_h_" + std::to_string(h) + "_k_"
        + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
72
73      Save(h, k, M, N, u, save_file_name);
74  }
75
76  ////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////
77
78  //public contstructor/s
79  ParDiffEqBurgersImplicit::ParDiffEqBurgersImplicit(double h, double k)
80      : h(h), k(k)
81  {
82      beta = ((0.5 * pow(h, 2)) / k) - 0.01; /*satisfy (beta*k)/h^2 < 0.5 for stability*/
83
84      MidPoint(h, M, midpoint);
85      IniMatrix(M, M, u);
86      IniVector(M, b);
87      InitialConditions(h, M, M, u, midpoint);
88  }
89  //public function/s
90  void ParDiffEqBurgersImplicit::Implicit()
91  {
92      const double s{ (beta * k) / pow(h,2) };
93      const double s_0{ -s };
94      const double s_1{ 1 + 2 * s };
95      const double s_2{ -s };;
96
97      for (size_t j = 0; j < M - 1; j++)
98      {
99          SetVectorb(j);
100         Tri solve(s_0, s_1, s_2, M, j, b, u);
101     }
102
103     std::string save_file_name{ "Results/Eq4/Implicit_Burgers_h_" + std::to_string(h) +
        "_k_" + std::to_string(k) + "_beta_" + std::to_string(beta) + ".csv" };
104
105     Save(h, k, M, M, u, save_file_name);
106 }
107 void ParDiffEqBurgersImplicit::SetVectorb(const size_t column)
108 {
109     b[0] = u[0][column];
110     for (size_t i = 1; i < M - 1; i++)
111     {
112         b[i] = u[i][column] - (k / (4 * h)) * (pow(u[i + 1][column], 2) - pow(u[i - 1][
            column], 2));
113     }
114     b[M - 1] = u[M - 1][column];
115 }
```

```cpp
//main.cpp

#include "0.0.Utility.h"
#include "1.Eq1.h"
#include "2.Eq2.h"
#include "3.Eq3.h"
#include "4.Burgers.h"

const static double h{ 0.1 };
const static double k{ 0.005 };

void DemoParDiffEq1()
{
    ParDiffEq1Euler eq1_eul(h, k);
    eq1_eul.Euler();

    ParDiffEq1RK2 eq1_rk2(h, k);
    eq1_rk2.RK2();
}
void DemoParDiffEq2()
{
    ParDiffEq2Euler eq2_eul(h, k);
    eq2_eul.Euler();

    ParDiffEq2RK2 eq2_rk2(h, k);
    eq2_rk2.RK2();
}
void DemoParDiffEq3()
{
    ParDiffEq3Euler eq3_eul(h, k);
    eq3_eul.Euler();

    ParDiffEq3RK2 eq3_rk2(h, k);
    eq3_rk2.RK2();

    ParDiffEq3Implicit eq3_implicit(h, k);
    eq3_implicit.Implicit();
}
void DemoBurgersEq()
{
    ParDiffEqBurgersEuler Burgers_eul(h, k);
    Burgers_eul.Euler();

    ParDiffEqBurgersRK2 Burgers_rk2(h, k);
    Burgers_rk2.RK2();

    ParDiffEqBurgersImplicit Burgers_implicit(h, k);
    Burgers_implicit.Implicit();
}

void MainMenu(bool& run)
{
    size_t choice{ 0 };
    const size_t final_choice{ 4 };

    system("cls");
    do
    {
        std::cout << "0. par_u/par_t + par_u/par_x = 0, PARABOLIC" << "\n";
        std::cout << "1. par_u/par_t + u*par_u/par_x = 0, PARABOLIC" << "\n";
        std::cout << "2. par_u/par_t - beta*par_u^2/par_x^2 = 0, beta > 0 => ELIPTIC,
        beta < 0 => HYPERBOLIC" << "\n";
        std::cout << "3. Burgers' equation: par_u/par_t + u*par_u/par_x -
        beta*par_u^2/par_x^2=0 = 0 , beta > 0 => ELIPTIC, beta < 0 => HYPERBOLIC" << "\n"
        ;
        std::cout << "4. Exit" << "\n";
        std::cin >> choice;
        if (choice > final_choice)
        {
```

```
67              std::cout << "Wrong choice!";
68              Sleep(1000);
69              system("cls");
70          }
71          system("cls");
72      } while (choice > final_choice);
73      if (choice == 0)
74      {
75          DemoParDiffEq1();
76      }
77      else if (choice == 1)
78      {
79          DemoParDiffEq2();
80      }
81      else if (choice == 2)
82      {
83          DemoParDiffEq3();
84      }
85      else if (choice == 3)
86      {
87          DemoBurgersEq();
88      }
89      else if (choice == final_choice)
90      {
91          run = false;
92      }
93      else
94      {
95          throw std::string("Fatal Error!");
96      }
97  }
98  int main()
99  {
100     bool run{ true };
101
102     while (run == true)
103     {
104         MainMenu(run);
105     }
106     return 0;
107 }
```