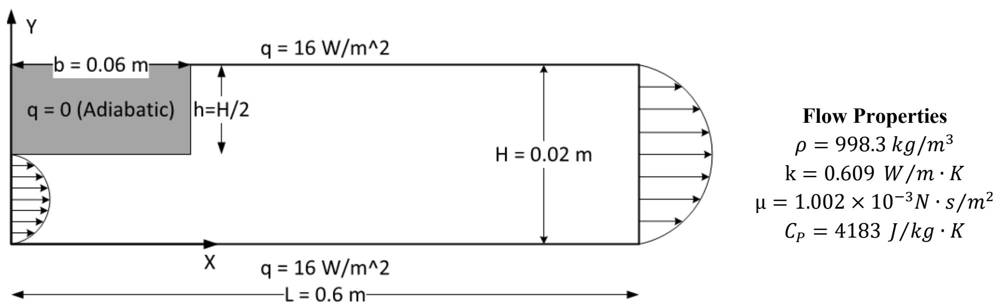
MEEN 644 - Final Exam

Logan Harbour
May 2, 2019

1 Problem statement



NOTE: The heat flux in the figure above for was changed to a cooled plate with a flux of $q = 64\ \text{W/m}^2$.

Consider an incompressible steady laminar flow over a backward facing step shown above. The properties of the fluid and geometric conditions are also given above. The bottom and top walls are maintained at constant heat flux boundary condition.

The inlet velocity is defined as

$$u(y) = u_{\max}\left(\frac{4y}{H}\right)\left(2 - \frac{4y}{H}\right), \quad \text{where } 0.01 > y > 0\,,$$

and the inlet temperature profile is defined as

$$\frac{T(y) - T_\mathrm{w}}{T_{\max} - T_\mathrm{w}} = \left(\frac{4y}{H}\right)\left(2 - \frac{4y}{H}\right), \quad \text{where } 0.01 > y > 0\,,$$

where $T_\mathrm{w} = 0, T_{\max} = 1.5$, and the Reynolds number is based on step height, i.e., $\mathrm{Re} = \rho u_{\max} h/\mu$.

Write a finite volume code to predict flow and temperature fields. Represent the solution to the 1-D convection-diffusion equation by the Power Law. Link velocity and pressure fields using the SIMPLE algorithm. Declare convergence when R_u, R_v, R_p, and $R_T \le 10^{-6}$.

In this exam, the following tasks are performed:

1. **(60 points)** Make calculations using Re = 200 using 160×30, 160×50, 160×70, and 160×90 CVs, respectively. Calculate reattachment length for each grid size and print in a tabular form. For the 160×90 CV case, plot the following figures:
   (a) Plot the temperature profile at $(x - b)/H = 6, 12$ and 24.
   (b) Plot both upper and lower wall temperature along the wall length.

(c) Plot Nusselt number for both upper and lower wall along the channel length.

2. **(40 points)** Calculate reattachment length for Re = 100, 300, and 400. Use 160×70 CVs. Compare your results with the experimental data in the reference. Print your comparison results in tabular form. For each Reynolds number, plot the u and v-velocity profiles at $(x-b)/H = 6, 12$, and 24.

2 Preliminaries

See Homeworks 4 and 5 for discussion of the solving methodology. The only difference between the methods used in said two homeworks was the implementation of variable material properties for k and μ. At initialization, these properties were set at the CV centroids as

$$k_p^{T_{i,j}} = \begin{cases} 10^{-99}\ \mathrm{W/m\cdot k} & x^{T_{i,j}} \le b \quad \text{and} \quad y^{T_{i,j}} \ge h \\ 0.609\ \mathrm{W/m\cdot k} & \text{otherwise} \end{cases}, \tag{1}$$

$$\mu_p^{u_{i,j}} = \begin{cases} 10^{99}\ \mathrm{N\cdot s/m^2} & x^{u_{i,j}} \le b \quad \text{and} \quad y^{u_{i,j}} \ge h \\ 1.002\times 10^{-3}\ \mathrm{N\cdot s/m^2} & \text{otherwise} \end{cases}, \tag{2}$$

$$\mu_p^{v_{i,j}} = \begin{cases} 10^{99}\ \mathrm{N\cdot s/m^2} & x^{v_{i,j}} \le b \quad \text{and} \quad y^{v_{i,j}} \ge h \\ 1.002\times 10^{-3}\ \mathrm{N\cdot s/m^2} & \text{otherwise} \end{cases}, \tag{3}$$

where $x^{\phi_{i,j}}$ represents the x-position of node $\phi_{i,j}$, and $y^{\phi_{i,j}}$ represents the y-position of node $\phi_{i,j}$.

The harmonic mean is then utilized to obtain the material properties at the CV edges. This is arbitrarily defined for material property m of variable ϕ as

$$m_n^{\phi_{i,j}} = \frac{2m_p^{\phi_{i,j}} m_p^{\phi_{i,j+1}}}{m_p^{\phi_{i,j}} + m_p^{\phi_{i,j+1}}}, \tag{4}$$

$$m_e^{\phi_{i,j}} = \frac{2m_p^{\phi_{i,j}} m_p^{\phi_{i+1,j}}}{m_p^{\phi_{i,j}} + m_p^{\phi_{i+1,j}}}, \tag{5}$$

$$m_s^{\phi_{i,j}} = \frac{2m_p^{\phi_{i,j}} m_p^{\phi_{i,j-1}}}{m_p^{\phi_{i,j}} + m_p^{\phi_{i,j-1}}}, \tag{6}$$

$$m_w^{\phi_{i,j}} = \frac{2m_p^{\phi_{i,j}} m_p^{\phi_{i-1,j}}}{m_p^{\phi_{i,j}} + m_p^{\phi_{i-1,j}}}. \tag{7}$$

The method of an arbitrarily large μ and arbitrarily small k within the step was utilize to implement the solid-fluid interface.

3 Results

For all the results that follow, the following experimental correlation from Goldstein is used as the "experimental" comparison, given that the experimental results are not given for a specific Re:

$$x_r = h \times (2.13 + 0.021\ \mathrm{Re}).$$

3.1 Problem 1: Re = 200

The results requested follow in Table 1 and Figures 1, 2, and 3.

Table 1: Comparison of reattachment lengths for each grid size with Re = 200.

Grid size	Numerical	Experimental	Difference
160×30	0.0612		3.31%
160×50	0.0617	0.0633	2.53%
160×70	0.0618		2.31%
160×90	0.0619		2.21%

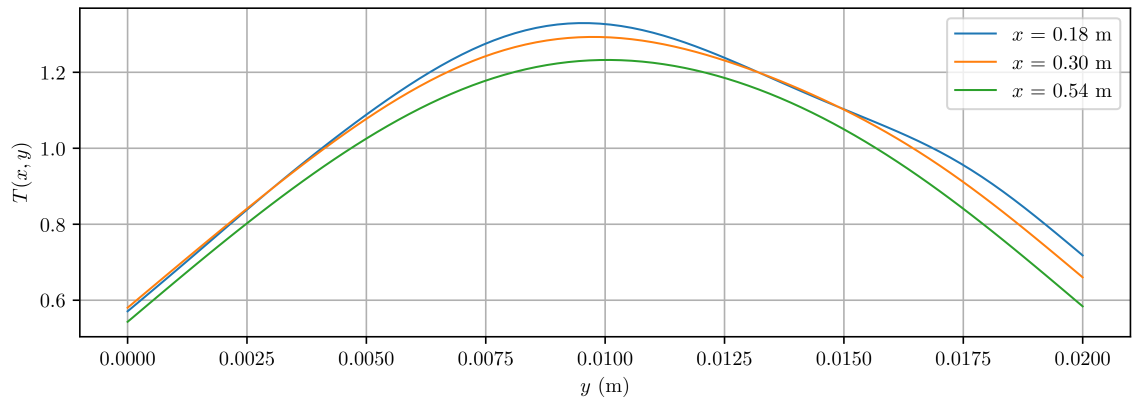


Figure 1: The y-temperature profile at various points in the channel for the 160×90 CV case with Re = 200.

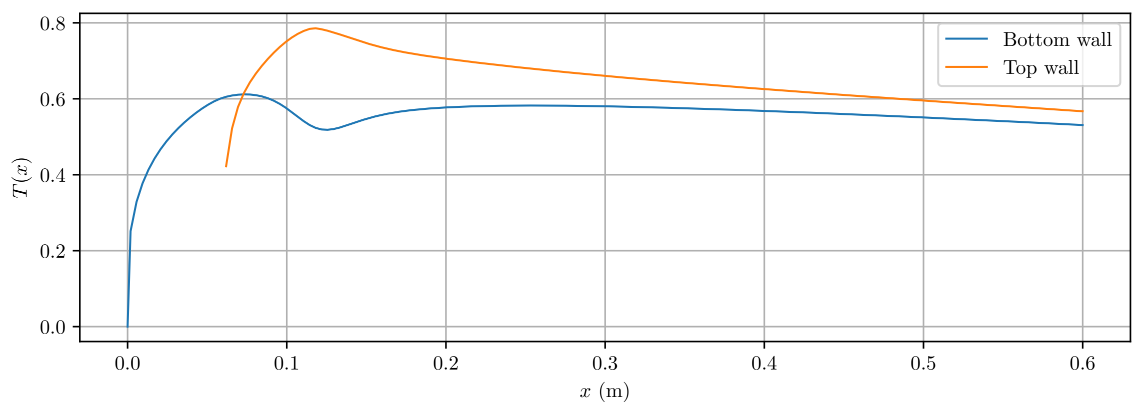


Figure 2: The wall temperature profiles for the 160×90 CV case with Re = 200. The final bottom wall temperature is 0.531 and the final top wall temperature is 0.567.

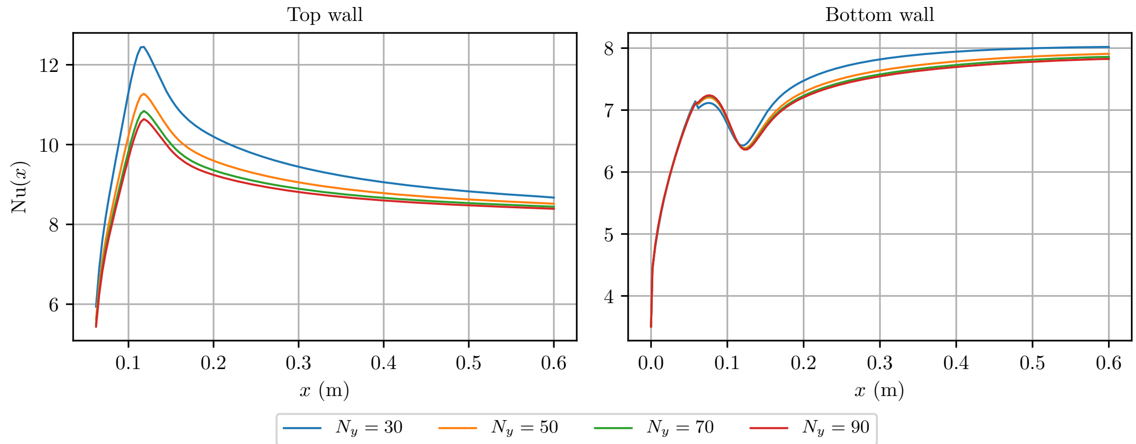


Figure 3: The wall Nusselt numbers for the 160×90 CV case with Re = 200. The final value for Nu with $N_y = 90$ for the bottom and top walls is 7.83 and 8.39, respectively.

General analysis for problem 1 is as follows:

- Figure 1 – The temperature profile is fully developed at the end of the channel, as evident by the fact that $T(0.54\text{ m}, y)$ reaches a maximum at the center of the channel ($y = 0.01$ m). In addition, we see minima a the walls for each case, as expected due to the prescribed outgoing heat flux and developed flow.
- Figure 2 – The top wall temperature profile reaches a maximum after the step as an effect of mixing after the step. Similarly, the bottom wall sees the opposite effect, as expected.
- Figure 3 – Increased mixing results in a higher Nusselt number after the step for the top wall and a lower Nusselt number after the step for the bottom wall, as expected.

3.2 Problem 2: Re = 100, 300, and 400

The results requested follow in Table 2 and Figures 4 and 5.

Table 2: Comparison of reattachment lengths with 160 × 70 CVs.

Re	Numerical	Experimental	Difference
100	0.0377	0.0432	10.80%
300	0.0792	0.0843	6.03%
400	0.0850	0.1053	19.32%

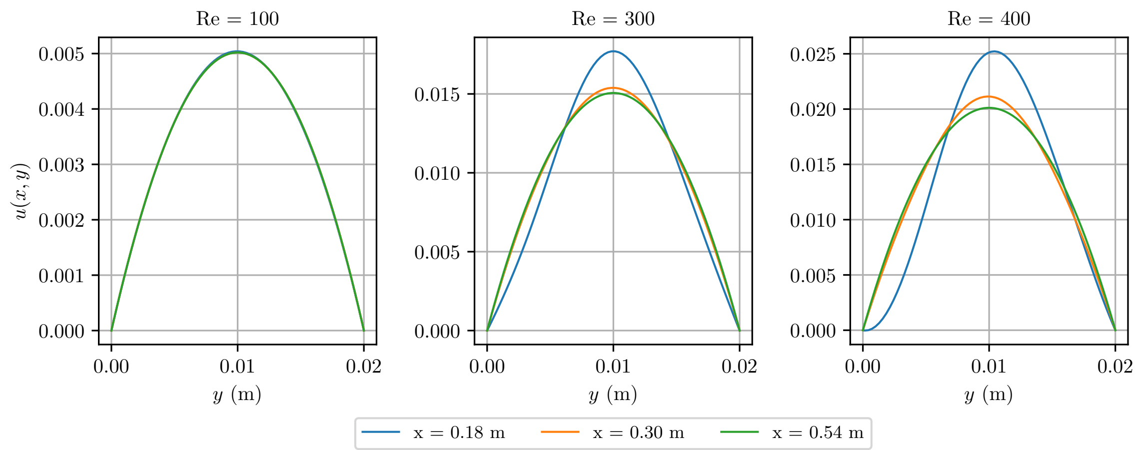


Figure 4: The u-velocity profiles at various points in the channel for the 160×70 CV case.

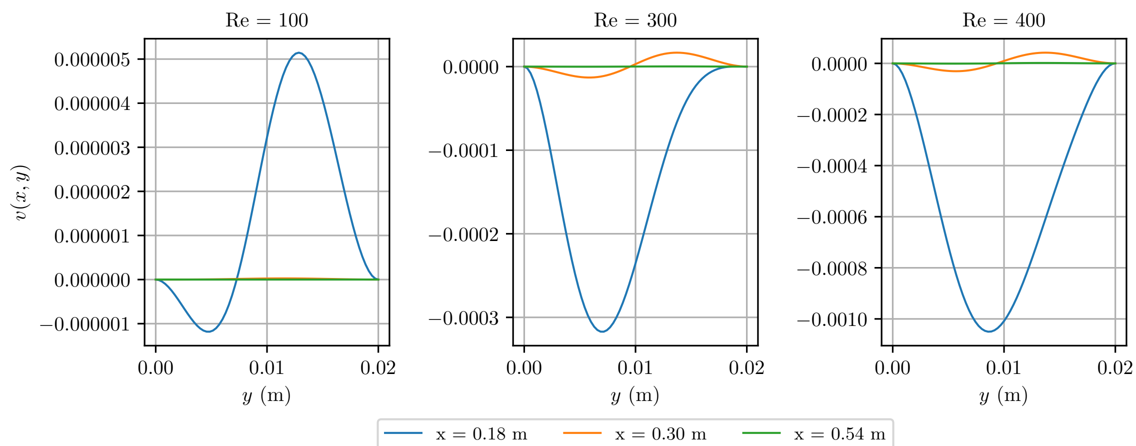


Figure 5: The v-velocity profiles at various points in the channel for the 160×70 CV case.

General analysis for problem 2 is as follows:

- Figure 4 – As expected, we see that with increasing Reynolds number, the flow takes longer to fully develop. For all cases, we see that the flow is fully developed by the end of the channel.
- Figure 5 – We see the effect of mixing after the step with the significantly negative velocities that occur at $x = 0.18$ m. As the flow develops, v as a function of x becomes flat, as expected.

Code listing

For the implementation, we have the following files:

- `Makefile` – Allows for compiling the c++ project with `make`.
- `final.cpp` – Contains the `main()` function that is required by C that runs the cases requested in this problem set.

- `Problem.h` – Contains the header for the `Problem` class which is the main driver for a `Flow2D::Problem`.
- `Variable.h` – Contains the `Flow2D::Variable` class, which is a storage container for a single variable (i.e., u).
- `Problem.cpp` – Contains the `run()` functions that executes a `Problem`.
- `Problem_coefficients.cpp` – Contains the functions for solving coefficients in a `Problem`.
- `Problem_corrections.cpp` – Contains the functions for correcting solutions in a `Problem`.
- `Problem_residuals.cpp` – Contains the functions for computing residuals in a `Problem`.
- `Problem_solvers.cpp` – Contains the functions for sweeping and solving in a `Problem`.
- `Matrix.h` – Contains the `Matrix` class which provides storage for a matrix with various standard matrix operations.
- `TriDiagonal.h` – Contains the `TriDiagonal` class which provides storage for a tri-diagonal matrix including the TDMA solver found in the member function `solveTDMA()`.
- `Vector.h` – Contains the `Vector` class for one-dimensional vector storage.
- `postprocess.py` - Produces the plots and tables in this report.

Makefile

```
src = $(wildcard *.cpp)
obj = $(src:.cpp=.o)
CXXFLAGS = -std=c++14
CCFLAGS = $(CXXFLAGS)

final-opt: $(obj)
	clang++ -o $@ $^

.PHONY: clean
clean:
	rm -f $(obj) final-opt
```

final.cpp

```
#include "Problem.h"

using namespace Flow2D;
using namespace std;

void
run(const double Re, const unsigned int Nx, const unsigned int Ny)
{
  const double Lx = 0.6;
  const double Ly = 0.02;
  const double cp = 4183;
  const double k = 0.609;
  const double rho = 998.3;
  const double mu = 0.001002;
  const double q_val = -64;
  const double Sx = 0.06;
  const double Sy = 0.5 * Ly;
  const double T_max = 1.5;
  const double T_w = 0;
  const double u_max = 2 * mu * Re / (rho * Ly);

  // Function to fill a material with v outside of the step and vs inside the step
  auto step_mat = [Sx, Sy](const double v, const double vs, const vector<double> p) -> double {
    // Inside of step
    if (p[0] <= Sx && p[1] >= Sy)
      return vs;
```

```cpp
    // Outside of step
    else
      return v;
  };

  // Function to fill u initial condition
  auto u_ic = [u_max, Ly, Sy, Ny](const vector<double> p) -> double {
    // Left side and from 0 < y < Ly / 2
    if ((p[0] == 0) && (p[1] < Sy))
      return u_max * (4 * p[1] / Ly) * (2 - (4 * p[1]) / Ly);
    // Zero otherwise
    else
      return 0;
  };

  // Function to fill v initial condition (all zero)
  auto v_ic = [](const vector<double> p) -> double { return 0; };

  // Function to fill T initial condition
  auto T_ic = [T_max, T_w, Ly, Sy, Ny](const vector<double> p) -> double {
    // Left side and from 0 < y < Ly / 2
    if ((p[0] == 0) && (p[1] < Sy))
      return (T_max - T_w) * (4 * p[1] / Ly) * (2 - (4 * p[1]) / Ly) + T_w;
    // Zero otherwise
    else
      return 0;
  };

  // Function to fill heat flux
  auto q = [q_val, Ly, Sx](const vector<double> p) -> double {
    // Bottom plate or top plate right of step
    if (p[1] == 0 || ((p[1] == Ly) && (p[0] > Sx)))
      return q_val;
    // Zero otherwise
    else
      return 0;
  };

  // Standard inputs
  InputArguments input;
  input.Lx = Lx;
  input.Ly = Ly;
  input.k = [step_mat, k](const vector<double> p) { return step_mat(k, 1E-99, p); };
  input.mu = [step_mat, mu](const vector<double> p) { return step_mat(mu, 1E99, p); };
  input.rho = rho;
  input.cp = cp;
  input.L_ref = Lx;
  input.q = q;
  input.u_ic = u_ic;
  input.v_ic = v_ic;
  input.T_ic = T_ic;
  input.u_ref = u_max;

  Problem problem(Nx, Ny, input);
  problem.run();

  const string prefix = "Re" + to_string((int)Re) + "_Nx" + to_string(Nx) + "_Ny" + to_string(Ny);
  problem.save(Variables::u, "results/" + prefix + "_u.csv");
  problem.save(Variables::v, "results/" + prefix + "_v.csv");
  problem.save(Variables::T, "results/" + prefix + "_T.csv");
}

int
main()
{
  // Problem 1
  for (const unsigned int Ny : {30, 50, 70, 90})
  {
    cout << "Problem 1: Re = 200, 160x" << to_string(Ny) << endl << "  ";
    run(200, 160, Ny);
  }

  // Problem 2
  for (const double Re : {100, 300, 400})
  {
    cout << "Problem 2 - Re = " << to_string((int)Re) << ", 160x60" << endl << "  ";
    run(Re, 160, 70);
  }
}
```

Problem.h

```cpp
#ifndef PROBLEM_H
#define PROBLEM_H

#include <cmath>
#include <ctime>
#include <functional>
#include <iomanip>
#include <iostream>
#include <map>

#include "Variable.h"

namespace Flow2D
{

using namespace std;

struct InputArguments
{
  double Lx, Ly;
  double L_ref, u_ref;
  function<double(const vector<double>)> u_ic, v_ic, T_ic, q;
  function<double(const vector<double>)> k, mu;
  double cp, rho;
  bool debug = false;
  double alpha_p = 0.7;
  double alpha_uv = 0.4;
  unsigned int max_main_its = 10000;
  unsigned int max_aux_its = 1000;
  double tol = 1.0e-6;
};

class Problem
{
public:
  Problem(const unsigned int Nx, const unsigned int Ny, const InputArguments & input);

  void run();

  // Public access to printing and saving variable results
  void print(const Variables var,
             const string prefix = "",
             const bool newline = false,
             const unsigned int pr = 5) const
  {
    variables.at(var).print(prefix, newline, pr);
  }
  void save(const Variables var, const string filename) const { variables.at(var).save(filename); }

private:
  void fillMaterial(Matrix<Coefficients> & m,
                    function<double(const vector<double> &)> func,
                    const Variable & var);

  // Problem_corrections.cpp
  void correctMain();
  void correctAux();
  void pCorrect();
  void pBCCorrect();
  void TBCCorrect();
  void uCorrect();
  void uBCCorrect();
  void vCorrect();

  // Problem_coefficients.cpp
  void fillCoefficients(const Variable & var);
  void pcCoefficients();
  void TCoefficients();
  void uCoefficients();
  void vCoefficients();
  void fillPowerLaw(Coefficients & a,
                    const Coefficients & D,
                    const Coefficients & F,
                    const double & b = 0);

  // Problem_residuals.cpp
  void computeMainResiduals();
```

```cpp
  void computeAuxResiduals();
  double pResidual() const;
  double TResidual() const;
  double velocityResidual(const Variable & var) const;

  // Problem_solvers.cpp
  void solveMain();
  void solveAux();
  void solve(Variable & var);
  void sweepColumns(Variable & var, const bool west_east = true);
  void sweepRows(Variable & var, const bool south_north = true);
  void sweepColumn(const unsigned int i, Variable & var);
  void sweepRow(const unsigned int j, Variable & var);
  void solveVelocities();

  // Quicker v^5 for velocityCoefficients() (yes, it's actually much faster...)
  static const double pow5(const double & v) { return v * v * v * v * v; }

protected:
  // Number of pressure CVs
  const unsigned int Nx, Ny;

  // Geometry [m]
  const double Lx, Ly, dx, dy;

  // Non-constant material properties
  Matrix<Coefficients> k, mu_u, mu_v;
  // Constant material properties
  const double rho, cp;
  // Residual references
  const double L_ref, u_ref;
  // Heat flux boundary condition
  function<double(const vector<double>)> q;

  // Enable debug mode (printing extra output)
  const bool debug;

  // Maximum iterations
  const unsigned int max_main_its, max_aux_its;
  // Iteration tolerance
  const double tol;
  // Pressure relaxation
  const double alpha_p;
  // Number of iterations completed
  unsigned int main_iterations = 0;
  unsigned int aux_iterations = 0;

  // Variables
  Variable u, v, pc, p, T;
  // Variable map
  map<const Variables, const Variable &> variables;

  // Whether or not we converged
  bool main_converged = false;
  bool converged = false;
  // Run start time
  clock_t start;
};

} // namespace Flow2D
#endif /* PROBLEM_H */
```

Variable.h

```cpp
#ifndef VARIABLE_H
#define VARIABLE_H

#include "Matrix.h"
#include "TriDiagonal.h"
#include "Vector.h"

#include <functional>

namespace Flow2D
{

using namespace std;

// Storage for coefficients for a single CV
struct Coefficients
{
  double p = 0, n = 0, e = 0, s = 0, w = 0, b = 0;
  void print(const unsigned int pr = 5) const
  {
    cout << setprecision(pr) << scientific << "n = " << n << ", e = " << e << ", s = " << s
         << ", w = " << w << ", p = " << p << ", b = " << b << endl;
  }
};

// Enum for variable types
enum Variables
{
  u,
  v,
  pc,
  p,
  T
};

// Conversion from variable type to its string
static string
VariableString(Variables var)
{
  switch (var)
  {
    case Variables::u:
      return "u";
    case Variables::v:
      return "v";
    case Variables::pc:
      return "pc";
    case Variables::p:
      return "p";
    case Variables::T:
      return "T";
  }
}

// General storage structure for primary and auxilary variables
struct Variable
{
  // Constructor for a primary variable
  Variable(const Variables name,
           const unsigned int Nx,
           const unsigned int Ny,
           const double dx,
           const double dy,
           const double alpha,
           function<double(const vector<double>)> ic = [](const vector<double> p) { return 0; })
    : name(name),
      string(VariableString(name)),
      Nx(Nx),
      Ny(Ny),
      dx(dx),
      dy(dy),
      Mx(Nx - 1),
      My(Ny - 1),
      w(1 / alpha),
      a(Nx, Ny),
      phi(Nx, Ny),
      Ax(Nx - 2),
```

```
    Ay(Ny - 2),
    bx(Nx - 2),
    by(Ny - 2)
{
  for (unsigned int i = 0; i <= Mx; ++i)
  {
    phi(i, 0) = ic(point(i, 0));
    phi(i, My) = ic(point(i, My));
  }
  for (unsigned int j = 0; j <= My; ++j)
  {
    phi(0, j) = ic(point(0, j));
    phi(Mx, j) = ic(point(Mx, j));
  }
}

// Constructor for an auxilary variable (no solver storage)
Variable(const Variables name,
         const unsigned int Nx,
         const unsigned int Ny,
         const double dx,
         const double dy,
         function<double(const vector<double>)> ic = [](const vector<double> p) { return 0; })
  : name(name),
    string(VariableString(name)),
    Nx(Nx),
    Ny(Ny),
    dx(dx),
    dy(dy),
    Mx(Nx - 1),
    My(Ny - 1),
    phi(Nx, Ny)
{
  for (unsigned int i = 0; i <= Mx; ++i)
  {
    phi(i, 0) = ic(point(i, 0));
    phi(i, My) = ic(point(i, My));
  }
  for (unsigned int j = 0; j <= My; ++j)
  {
    phi(0, j) = ic(point(0, j));
    phi(Mx, j) = ic(point(Mx, j));
  }
}

// Solution matrix operations
void operator=(const double v) { phi = v; }
const double & operator()(const unsigned int i, const unsigned int j) const { return phi(i, j); }
double & operator()(const unsigned int i, const unsigned int j) { return phi(i, j); }
void print(const string prefix = "", const bool newline = false, const unsigned int pr = 5) const
{
  phi.print(prefix, newline, pr);
}
void save(const string filename) const { phi.save(filename); }
void reset() { phi = 0; }

// Get the point in space associated with an i, j for this CV
const vector<double> point(const unsigned int i, const unsigned int j) const
{
  const double id = (double)i, jd = (double)j;
  vector<double> pt = {dx * (i != 0) * (id - 0.5 * (1 + (i == Mx))),
                       dy * (j != 0) * (jd - 0.5 * (1 + (j == My)))};
  if (name == Variables::u)
    pt[0] = id * dx;
  if (name == Variables::v)
    pt[1] = jd * dy;
  return pt;
}

// Coefficient debug
void printCoefficients(const string prefix = "",
                       const bool newline = false,
                       const unsigned int pr = 5) const
{
  for (unsigned int i = 1; i < Nx - 1; ++i)
    for (unsigned int j = 1; j < Ny - 1; ++j)
    {
      cout << prefix << "(" << i << ", " << j << "): ";
      a(i, j).print(pr);
    }
  if (newline)
```

```
        cout << endl;
    }

    // Variable enum name
    const Variables name;
    // Variable string
    const string string;
    // Variable size
    const unsigned int Nx, Ny;
    // Mesh size
    const double dx, dy;
    // Maximum variable index that is being solved
    const unsigned int Mx, My;
    // Relaxation coefficient used in solving linear systems
    const double w = 0;
    // Matrix coefficients
    Matrix<Coefficients> a;
    // Variable solution
    Matrix<double> phi;
    // Linear system LHS for both sweep directions
    TriDiagonal Ax, Ay;
    // Linear system RHS for both sweep directions
    Vector<double> bx, by;
}; // namespace Flow2D

} // namespace Flow2D
#endif /* VARIABLE_H */
```

Problem.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

Problem::Problem(const unsigned int Nx,
                 const unsigned int Ny,
                 const InputArguments & input)
  : // Number of pressure CVs
    Nx(Nx),
    Ny(Ny),
    // Domain sizes
    Lx(input.Lx),
    Ly(input.Ly),
    dx(Lx / Nx),
    dy(Ly / Ny),
    // Residual references
    L_ref(input.L_ref),
    u_ref(input.u_ref),
    // Heat flux boundary condition
    q(input.q),
    // Initialize material properties
    k(Nx + 1, Ny + 1),
    mu_u(Nx, Ny + 1),
    mu_v(Nx + 1, Ny),
    rho(input.rho),
    cp(input.cp),
    // Enable debug
    debug(input.debug),
    // Solver properties
    max_main_its(input.max_main_its),
    max_aux_its(input.max_aux_its),
    tol(input.tol),
    alpha_p(input.alpha_p),
    // Initialize variables for u, v, pc (solved variables)
    u(Variables::u, Nx + 1, Ny + 2, dx, dy, input.alpha_uv, input.u_ic),
    v(Variables::v, Nx + 2, Ny + 1, dx, dy, input.alpha_uv, input.v_ic),
    pc(Variables::pc, Nx + 2, Ny + 2, dx, dy, 1),
    T(Variables::T, Nx + 2, Ny + 2, dx, dy, 1, input.T_ic),
    // Initialize aux variables
    p(Variables::p, Nx + 2, Ny + 2, dx, dy)
{
  // Add into variable map for access outside of class
  variables.emplace(Variables::u, u);
  variables.emplace(Variables::v, v);
  variables.emplace(Variables::pc, pc);
  variables.emplace(Variables::p, p);
  variables.emplace(Variables::T, T);

  // Fill non-constant materials
  fillMaterial(k, input.k, T);
  fillMaterial(mu_u, input.mu, u);
  fillMaterial(mu_v, input.mu, v);
}

void
Problem::run()
{
  // Store start time
  start = clock();

  // Solve main variables
  for (unsigned int l = 0; l < max_main_its; ++l)
  {
    solveMain();
    correctMain();
    computeMainResiduals();

    // Break out if we've converged
    if (main_converged)
      break;
  }

  // Ensure main variables converged
  if (!main_converged)
    cout << "Main variables did not converge after " << max_main_its << " iterations!" << endl;
```

```
  // Solve aux variables
  for (unsigned int l = 0; l < max_aux_its; ++l)
  {
    solveAux();
    correctAux();
    computeAuxResiduals();

    // Exit if everything is converged
    if (converged)
      return;
  }

  // Oops. Didn't converge
  cout << "Aux variables did not converge after " << max_aux_its << " iterations!" << endl;
}

void
Problem::fillMaterial(Matrix<Coefficients> & m,
                      std::function<double(const std::vector<double> &)> func,
                      const Variable & var)
{
  // First, fill the variable everywhere (p, n, e, s, w)
  for (unsigned int i = 1; i < var.Mx; ++i)
    for (unsigned int j = 1; j < var.My; ++j)
      m(i, j).p = func(var.point(i, j));

  // And now fill with the harmonic mean at the interior edges
  for (unsigned int i = 1; i < var.Mx; ++i)
    for (unsigned int j = 1; j < var.My; ++j)
    {
      if (j != var.My - 1)
        m(i, j).n = 2 * m(i, j).p * m(i, j + 1).p / (m(i, j).p + m(i, j + 1).p);
      else
        m(i, j).n = m(i, j).p;
      if (i != var.Mx - 1)
        m(i, j).e = 2 * m(i, j).p * m(i + 1, j).p / (m(i, j).p + m(i + 1, j).p);
      else
        m(i, j).e = m(i, j).p;
      if (j != 1)
        m(i, j).s = 2 * m(i, j).p * m(i, j - 1).p / (m(i, j).p + m(i, j - 1).p);
      else
        m(i, j).s = m(i, j).p;
      if (i != 1)
        m(i, j).w = 2 * m(i, j).p * m(i - 1, j).p / (m(i, j).p + m(i - 1, j).p);
      else
        m(i, j).w = m(i, j).p;
    }
}
} // namespace Flow2D
```

Problem_coefficients.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::fillCoefficients(const Variable & var)
{
  if (var.name == Variables::pc)
    pcCoefficients();
  else if (var.name == Variables::T)
    TCoefficients();
  else if (var.name == Variables::u)
    uCoefficients();
  else if (var.name == Variables::v)
    vCoefficients();

  if (debug)
  {
    cout << var.string << " coefficients: " << endl;
    var.printCoefficients(var.string, true);
  }
}

void
Problem::pcCoefficients()
{
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
    {
      Coefficients & a = pc.a(i, j);

      if (i != 1)
        a.w = rho * dy * dy / u.a(i - 1, j).p;
      if (i != pc.Mx - 1)
        a.e = rho * dy * dy / u.a(i, j).p;
      if (j != 1)
        a.s = rho * dx * dx / v.a(i, j - 1).p;
      if (j != pc.My - 1)
        a.n = rho * dx * dx / v.a(i, j).p;
      a.p = a.n + a.e + a.s + a.w;
      a.b = rho * (dy * (u(i - 1, j) - u(i, j)) + dx * (v(i, j - 1) - v(i, j)));
    }
}

void
Problem::TCoefficients()
{
  for (unsigned int i = 1; i < T.Mx; ++i)
    for (unsigned int j = 1; j < T.My; ++j)
    {
      Coefficients D, F;

      // Diffusion coefficient
      D.n = k(i, j).n * dx / dy * (j == T.My - 1 ? 2.0 : 1.0);
      D.e = k(i, j).e * dy / dx * (i == T.Mx - 1 ? 2.0 : 1.0);
      D.s = k(i, j).s * dx / dy * (j == 1 ? 2.0 : 1.0);
      D.w = k(i, j).w * dy / dx * (i == 1 ? 2.0 : 1.0);

      // Heat flows
      F.n = dx * cp * rho * v(i, j);
      F.e = dy * cp * rho * u(i, j);
      F.s = dx * cp * rho * v(i, j - 1);
      F.w = dy * cp * rho * u(i - 1, j);

      // Compute and store power law coefficients
      fillPowerLaw(T.a(i, j), D, F);
    }
}

void
Problem::uCoefficients()
{
  for (unsigned int i = 1; i < u.Mx; ++i)
    for (unsigned int j = 1; j < u.My; ++j)
    {
      Coefficients D, F;
```

```
      // Width of the cell
      const double W = dx * (i == 1 || i == u.Mx - 1 ? 1.5 : 1.0);
      // North/south distances to pressure nodes
      const double dy_pn = dy * (j == u.My - 1 ? 0.5 : 1.0);
      const double dy_ps = dy * (j == 1 ? 0.5 : 1.0);

      // Diffusion coefficients
      D.n = mu_u(i, j).n * W / dy_pn;
      D.e = mu_u(i, j).e * dy / dx;
      D.s = mu_u(i, j).s * W / dy_ps;
      D.w = mu_u(i, j).w * dy / dx;

      // East and west flows
      F.e = rho * dy * (i == u.Mx - 1 ? u(u.Mx, j) : 0.5 * (u(i + 1, j) + u(i, j)));
      F.w = rho * dy * (i == 1 ? u(0, j) : 0.5 * (u(i - 1, j) + u(i, j)));

      // North and south flows
      if (i == 1) // Left boundary
      {
        F.n = rho * W * (v(0, j) + 3.0 * v(1, j) + 2.0 * v(2, j)) / 6.0;
        F.s = rho * W * (v(0, j - 1) + 3.0 * v(1, j - 1) + 2.0 * v(2, j - 1)) / 6.0;
      }
      else if (i == u.Mx - 1) // Right boundary
      {
        F.n = rho * W * (2.0 * v(i, j) + 3.0 * v(i + 1, j) + v(i + 2, j)) / 6.0;
        F.s = rho * W * (2.0 * v(i, j - 1) + 3.0 * v(i + 1, j - 1) + v(i + 2, j - 1)) / 6.0;
      }
      else // Interior (not left or right boundary)
      {
        F.n = rho * W * 0.5 * (v(i, j) + v(i + 1, j));
        F.s = rho * W * 0.5 * (v(i, j - 1) + v(i + 1, j - 1));
      }

      // Pressure RHS
      const double b = dy * (p(i, j) - p(i + 1, j));

      // Compute and store power law coefficients
      fillPowerLaw(u.a(i, j), D, F, b);

      // Explicitly set outflow condition
      if (i == u.Mx - 1)
      {
        u.a(i, j).p -= u.a(i, j).e;
        u.a(i, j).e = 0;
      }
    }
}

void
Problem::vCoefficients()
{
  for (unsigned int i = 1; i < v.Mx; ++i)
    for (unsigned int j = 1; j < v.My; ++j)
    {
      Coefficients D, F;

      // Height of the cell
      const double H = dy * (j == 1 || j == v.My - 1 ? 1.5 : 1.0);
      // East/west distances to pressure nodes
      const double dx_pe = dx * (i == v.Mx - 1 ? 0.5 : 1.0);
      const double dx_pw = dx * (i == 1 ? 0.5 : 1.0);

      // Diffusion coefficient
      D.n = mu_v(i, j).n * dx / dy;
      D.e = mu_v(i, j).e * H / dx_pe;
      D.s = mu_v(i, j).s * dx / dy;
      D.w = mu_v(i, j).w * H / dx_pw;

      // North and east flows
      F.n = rho * dx * (j == v.My - 1 ? v(i, v.My) : 0.5 * (v(i, j + 1) + v(i, j)));
      F.s = rho * dx * (j == 1 ? v(i, 0) : 0.5 * (v(i, j - 1) + v(i, j)));
      // East and west flows
      if (j == 1) // Bottom boundary
      {
        F.e = rho * H * (u(i, 0) + 3.0 * u(i, 1) + 2.0 * u(i, 2)) / 6.0;
        F.w = rho * H * (u(i - 1, 0) + 3.0 * u(i - 1, 1) + 2.0 * u(i - 1, 2)) / 6.0;
      }
      else if (j == v.My - 1) // Top boundary
      {
        F.e = rho * H * (2.0 * u(i, j) + 3.0 * u(i, j + 1) + u(i, j + 2)) / 6.0;
```

```cpp
        F.w = rho * H * (2.0 * u(i - 1, j) + 3.0 * u(i - 1, j + 1) + u(i - 1, j + 2)) / 6.0;
      }
      else // Interior (not top or bottom boundary)
      {
        F.e = rho * H * 0.5 * (u(i, j) + u(i, j + 1));
        F.w = rho * H * 0.5 * (u(i - 1, j) + u(i - 1, j + 1));
      }

      // Pressure RHS
      const double b = dx * (p(i, j) - p(i, j + 1));

      // Compute and store power law coefficients
      fillPowerLaw(v.a(i, j), D, F, b);
    }
}

void
Problem::fillPowerLaw(Coefficients & a,
                      const Coefficients & D,
                      const Coefficients & F,
                      const double & b)
{
  a.n = D.n * fmax(0, pow5(1 - 0.1 * fabs(F.n / D.n))) + fmax(-F.n, 0);
  a.e = D.e * fmax(0, pow5(1 - 0.1 * fabs(F.e / D.e))) + fmax(-F.e, 0);
  a.s = D.s * fmax(0, pow5(1 - 0.1 * fabs(F.s / D.s))) + fmax(F.s, 0);
  a.w = D.w * fmax(0, pow5(1 - 0.1 * fabs(F.w / D.w))) + fmax(F.w, 0);
  a.p = a.n + a.e + a.s + a.w;
  a.b = b;
}

} // namespace Flow2D
```

Problem_corrections.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::correctMain()
{
  uCorrect();
  vCorrect();
  pCorrect();
  pBCCorrect();
  uBCCorrect();
}

void
Problem::correctAux()
{
  TBCCorrect();
}

void
Problem::pCorrect()
{
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
      p(i, j) += alpha_p * pc(i, j);

  // Set pressure correction back to zero
  pc.reset();

  if (debug)
    p.print("p corrected = ", true);
}

void
Problem::pBCCorrect()
{
  // Apply the edge values as velocity is set
  for (unsigned int i = 0; i <= pc.Mx; ++i)
  {
    p(i, 0) = p(i, 1);
    p(i, pc.My) = p(i, pc.My - 1);
  }
  for (unsigned int j = 0; j <= pc.My; ++j)
  {
    p(0, j) = p(1, j);
    p(pc.Mx, j) = p(pc.Mx - 1, j);
  }

  if (debug)
    p.print("p boundary condition corrected = ", true);
}
void
Problem::TBCCorrect()
{
  for (unsigned int j = 0; j <= T.My; ++j)
    T(T.Mx, j) = (3 * T(T.Mx - 1, j) - T(T.Mx - 2, j)) / 2;

  for (unsigned int i = 1; i < T.Mx; ++i)
  {
    T(i, 0) = T(i, 1) + q(T.point(i, 0)) * dy / (2 * k(i, 1).p);
    T(i, T.My) = T(i, T.My - 1) + q(T.point(i, T.My)) * dy / (2 * k(i, T.My - 1).p);
  }
}

void
Problem::uCorrect()
{
  for (unsigned int i = 1; i < u.Mx; ++i)
    for (unsigned int j = 1; j < u.My; ++j)
      u(i, j) += dy * (pc(i, j) - pc(i + 1, j)) / u.a(i, j).p;

  if (debug)
    u.print("u corrected = ", true);
}
```

```
void
Problem::uBCCorrect()
{
  double m_in = 0, m_out = 0;
  for (unsigned int j = 1; j < u.My; ++j)
  {
    m_in += rho * dy * u(0, j);
    m_out += rho * dy * u(u.Mx - 1, j);
  }
  for (unsigned int j = 0; j <= u.My; ++j)
    u(u.Mx, j) = m_in * u(u.Mx - 1, j) / m_out;

  if (debug)
    u.print("u boundary condition corrected = ", true);
}

void
Problem::vCorrect()
{
  for (unsigned int i = 1; i < v.Mx; ++i)
    for (unsigned int j = 1; j < v.My; ++j)
      v(i, j) += dx * (pc(i, j) - pc(i, j + 1)) / v.a(i, j).p;

  if (debug)
    v.print("v corrected = ", true);
}

} // namespace Flow2D
```

Problem_residuals.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::computeMainResiduals()
{
  const double Rp = pResidual();
  const double Ru = velocityResidual(u);
  const double Rv = velocityResidual(v);

  if (Ru < tol && Rv < tol && Rp < tol)
  {
    if (debug)
      cout << "Main variables converged in " << main_iterations << " iterations" << endl;
    main_converged = true;
  }
}

void
Problem::computeAuxResiduals()
{
  double RT = TResidual();

  // Still not converged
  if (RT > tol)
    return;

  // Converged, finish up
  converged = true;

  // Print the result
  const double Rp = pResidual();
  const double Ru = velocityResidual(u);
  const double Rv = velocityResidual(v);
  cout << "Converged in " << noshowpos << fixed << setprecision(3)
       << 1.0 * (clock() - start) / CLOCKS_PER_SEC << " sec in " << main_iterations << " main and "
       << aux_iterations << " aux iterations: ";
  cout << noshowpos << setprecision(1) << scientific;
  cout << "p = " << Rp;
  cout << ", u = " << Ru;
  cout << ", v = " << Rv;
  cout << ", T = " << RT << endl;
}

double
Problem::pResidual() const
{
  double numer = 0;
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
      numer += abs(dy * (u(i - 1, j) - u(i, j)) + dx * (v(i, j - 1) - v(i, j)));
  return numer / (u_ref * L_ref);
}

double
Problem::TResidual() const
{
  double numer, numer_temp, denom = 0;
  for (unsigned int i = 1; i < T.Mx; ++i)
    for (unsigned int j = 1; j < T.My; ++j)
    {
      const Coefficients & a = T.a(i, j);
      numer_temp = a.p * T(i, j);
      denom += abs(numer_temp);
      numer_temp -= a.n * T(i, j + 1) + a.e * T(i + 1, j);
      numer_temp -= a.s * T(i, j - 1) + a.w * T(i - 1, j);
      numer += abs(numer_temp);
    }
  return numer / denom;
}

double
Problem::velocityResidual(const Variable & var) const
{
  double numer, numer_temp, denom = 0;
```

```
  for (unsigned int i = 1; i < var.Mx; ++i)
    for (unsigned int j = 1; j < var.My; ++j)
    {
      const Coefficients & a = var.a(i, j);
      numer_temp = a.p * var(i, j);
      denom += abs(numer_temp);
      numer_temp -= a.n * var(i, j + 1) + a.e * var(i + 1, j);
      numer_temp -= a.s * var(i, j - 1) + a.w * var(i - 1, j) + a.b;
      numer += abs(numer_temp);
    }
  return numer / denom;
}

} // namespace Flow2D
```

Problem_solvers.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::solveMain()
{
  ++main_iterations;
  if (debug)
    cout << endl << "Main iteration " << main_iterations << endl << endl;

  solve(u);
  solve(v);
  solve(pc);
}

void
Problem::solveAux()
{
  ++aux_iterations;
  if (debug)
    cout << endl << "Aux iteration " << aux_iterations << endl << endl;

  solve(T);
}

void
Problem::solve(Variable & var)
{
  if (debug)
    cout << "Solving variable " << var.string << endl << endl;

  // Fill the coefficients
  fillCoefficients(var);

  // Solve west to east
  sweepColumns(var);
  // Solve south to north
  sweepRows(var);
  // Solve east to west
  sweepColumns(var, false);

  if (debug)
    var.print(var.string + " sweep solution = ", true);
}

void
Problem::sweepRows(Variable & var, const bool south_north)
{
  if (debug)
    cout << "Sweeping " << var.string << (south_north ? " south to north" : " north to south")
         << endl;

  // Sweep south to north
  if (south_north)
    for (int j = 1; j < var.My; ++j)
      sweepRow(j, var);
  // Sweep north to south
  else
    for (int j = var.My - 1; j > 0; --j)
      sweepRow(j, var);
}

void
Problem::sweepColumns(Variable & var, const bool west_east)
{
  if (debug)
    cout << "Sweeping " << var.string << (west_east ? " east to west" : " west to east") << endl;

  // Sweep west to east
  if (west_east)
    for (int i = 1; i < var.Mx; ++i)
      sweepColumn(i, var);
  // Sweep east to west
  else
    for (int i = var.Mx - 1; i > 0; --i)
```

```
      sweepColumn(i, var);
}

void
Problem::sweepColumn(const unsigned int i, Variable & var)
{
  if (debug)
    cout << "Solving " << var.string << " column " << i << endl;

  auto & A = var.Ay;
  auto & b = var.by;

  // Fill for each cell
  for (unsigned int j = 1; j < var.My; ++j)
  {
    const Coefficients & a = var.a(i, j);
    b[j - 1] = a.b + a.w * var(i - 1, j) + a.e * var(i + 1, j);
    if (var.w != 1)
      b[j - 1] += a.p * var(i, j) * (var.w - 1);
    if (j == 1)
    {
      A.setTopRow(a.p * var.w, -a.n);
      if (var.name != Variables::pc)
        b[j - 1] += a.s * var(i, j - 1);
    }
    else if (j == var.My - 1)
    {
      A.setBottomRow(-a.s, a.p * var.w);
      if (var.name != Variables::pc)
        b[j - 1] += a.n * var(i, j + 1);
    }
    else
      A.setMiddleRow(j - 1, -a.s, a.p * var.w, -a.n);
  }

  if (debug)
  {
    A.print("A =");
    b.print("b =");
  }

  // Solve
  A.TDMA(b);

  if (debug)
    b.print("sol =", true);

  // Store solution
  for (unsigned int j = 1; j < var.My; ++j)
    var(i, j) = b[j - 1];
}

void
Problem::sweepRow(const unsigned int j, Variable & var)
{
  if (debug)
    cout << "Solving " << var.string << " row " << j << endl;

  auto & A = var.Ax;
  auto & b = var.bx;

  // Fill for each cell
  for (unsigned int i = 1; i < var.Mx; ++i)
  {
    const Coefficients & a = var.a(i, j);
    b[i - 1] = a.b + a.s * var(i, j - 1) + a.n * var(i, j + 1);
    if (var.w != 1)
      b[i - 1] += a.p * var(i, j) * (var.w - 1);
    if (i == 1)
    {
      A.setTopRow(a.p * var.w, -a.e);
      if (var.name != Variables::pc)
        b[i - 1] += a.w * var(i - 1, j);
    }
    else if (i == var.Mx - 1)
    {
      A.setBottomRow(-a.w, a.p * var.w);
      if (var.name != Variables::pc)
        b[i - 1] += a.e * var(i + 1, j);
    }
    else
```

```
      A.setMiddleRow(i - 1, -a.w, a.p * var.w, -a.e);
  }

  if (debug)
  {
    A.print("A =");
    b.print("b =");
  }

  // Solve
  A.TDMA(b);

  if (debug)
    b.print("sol =", true);

  // Store solution
  for (unsigned int i = 1; i < var.Mx; ++i)
    var(i, j) = b[i - 1];
}

} // namespace Flow2D
```

Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#define NDEBUG
#include <cassert>
#include <fstream>
#include <vector>

using namespace std;

/**
 * Class that holds a N x M matrix with common matrix operations.
 */
template <typename T>
class Matrix
{
public:
  Matrix() {}
  Matrix(const unsigned int N, const unsigned int M) : N(N), M(M), A(N, vector<T>(M)) {}

  // Const operator for getting the (i, j) element
  const T & operator()(const unsigned int i, const unsigned int j) const
  {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for getting the (i, j) element
  T & operator()(const unsigned int i, const unsigned int j)
  {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for setting the entire matrix to a value
  void operator=(const T v)
  {
    for (unsigned int j = 0; j < M; ++j)
      setRow(j, v);
  }

  // Prints the matrix
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 5) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int j = 0; j < M; ++j)
    {
      for (unsigned int i = 0; i < N; ++i)
        cout << showpos << scientific << setprecision(pr) << A[i][j] << " ";
      cout << endl;
    }
    if (newline)
      cout << endl;
  }
  // Saves the matrix in csv format
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int j = 0; j < M; ++j)
    {
      for (unsigned int i = 0; i < N; ++i)
      {
        if (i > 0)
          f << ",";
        f << setprecision(pr) << A[i][j];
      }
      f << endl;
    }
    f.close();
  }

  // Set the j-th row to v
  void setRow(const unsigned int j, const T v)
  {
    assert(j < M);
    for (unsigned int i = 0; i < N; ++i)
      A[i][j] = v;
```

```
  }
  // Set the i-th column to v
  void setColumn(const unsigned int i, const T v)
  {
    assert(i < N);
    for (unsigned int j = 0; j < M; ++j)
      A[i][j] = v;
  }

private:
  // The size of this matrix
  const unsigned int N = 0, M = 0;

  // Matrix storage
  vector<vector<T>> A;
};

#endif /* MATRIX_H */
```

TriDiagonal.h

```
#ifndef TRIDIAGONAL_H
#define TRIDIAGONAL_H

#define NDEBUG
#include <cassert>
#include <fstream>
#include "Vector.h"

using namespace std;

/**
 * Class that holds a tri-diagonal matrix and is able to perform TDMA in place
 * with a given RHS.
 */
class TriDiagonal
{
public:
  TriDiagonal() {}
  TriDiagonal(const unsigned int N, const double v = 0) : N(N), A(N, v), B(N, v), C(N - 1, v) {}

  // Setters for the top, middle, and bottom rows
  void setTopRow(const double b, const double c)
  {
    B[0] = b;
    C[0] = c;
  }
  void setMiddleRow(const unsigned int i, const double a, const double b, const double c)
  {
    assert(i < N - 1 && i != 0);
    A[i] = a;
    B[i] = b;
    C[i] = c;
  }
  void setBottomRow(const double a, const double b)
  {
    A[N - 1] = a;
    B[N - 1] = b;
  }

  // Prints the matrix
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 6) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int i = 0; i < N; ++i)
      cout << showpos << scientific << setprecision(pr) << (i > 0 ? A[i] : 0) << " " << B[i] << " "
           << (i < N - 1 ? C[i] : 0) << endl;
    if (newline)
      cout << endl;
  }
  // Saves the matrix in csv format
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int i = 0; i < N; ++i)
    {
      if (i > 0)
        f << setprecision(pr) << A[i] << ",";
      else
        f << "0"
          << ",";
      f << setprecision(pr) << B[i] << ",";
      if (i != N - 1)
        f << setprecision(pr) << C[i] << endl;
      else
        f << 0 << endl;
    }
    f.close();
  }

  void TDMA(Vector<double> & d)
  {
    // Forward sweep
    double tmp = 0;
    for (unsigned int i = 1; i < N; ++i)
    {
```

```
      tmp = A[i] / B[i - 1];
      B[i] -= tmp * C[i - 1];
      d[i] -= tmp * d[i - 1];
    }

    // Backward sweep
    d[N - 1] /= B[N - 1];
    for (unsigned int i = N - 2; i != numeric_limits<unsigned int>::max(); --i)
    {
      d[i] -= C[i] * d[i + 1];
      d[i] /= B[i];
    }
  }

protected:
  // Matrix size (N x N)
  unsigned int N = 0;

  // Left/main/right diagonal storage
  vector<double> A, B, C;
};

#endif /* TRIDIAGONAL_H */
```

Vector.h

```cpp
#ifndef VECTOR_H
#define VECTOR_H

#define NDEBUG
#include <cassert>
#include <fstream>
#include <vector>

using namespace std;

/**
 * Class that stores a 1D vector and enables printing and saving.
 */
template <typename T>
class Vector
{
public:
  Vector(const unsigned int N) : v(N), N(N) {}
  Vector() {}

  const T & operator()(const unsigned int i) const
  {
    assert(i < N);
    return v[i];
  }
  T & operator()(const unsigned int i)
  {
    assert(i < N);
    return v[i];
  }
  const T & operator[](const unsigned int i) const
  {
    assert(i < N);
    return v[i];
  }
  T & operator[](const unsigned int i)
  {
    assert(i < N);
    return v[i];
  }

  // Prints the vector
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 6) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int i = 0; i < v.size(); ++i)
      cout << showpos << scientific << setprecision(pr) << v[i] << " ";
    cout << endl;
    if (newline)
      cout << endl;
  }

  // Saves the vector
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int i = 0; i < v.size(); ++i)
      f << scientific << v[i] << endl;
    f.close();
  }

private:
  vector<T> v;
  const unsigned int N = 0;
};

#endif /* VECTOR_H */
```

postprocess.py

```
import numpy as np
import matplotlib.pyplot as plt
import glob
import os

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

Lx = 0.6
Ly = 0.02
b = 0.06
q = -64
rho = 998.3
k = 0.609
cp = 4183
mu = 0.001002
h = Ly / 2

x_r = lambda Re: (2.13 + 0.021 * Re) * h

################################################################################
# Problem 1

Ny_vals = [30, 50, 70, 90]
Nx = 160
dx = Lx / Nx

# Reattachment lengths, u sampled at T points
xr, T, Nu_top, Nu_bot = {}, {}, {}, {}
for Ny in Ny_vals:
    dy = Ly / Ny
    u = np.loadtxt('results/Re200_Nx160_Ny{}_u.csv'.format(Ny), delimiter=',').T
    T[Ny] = np.loadtxt('results/Re200_Nx160_Ny{}_T.csv'.format(Ny), delimiter=',').T
    # u at T points
    uT = np.zeros(T[Ny].shape)
    for i in range(1, T[Ny].shape[0] - 1):
        for j in range(u.shape[1]):
            uT[0, j] = u[0, j]
            uT[u.shape[0], j] = u[u.shape[0] - 1, j]
            uT[i, j] = (u[i - 1, j] + u[i, j]) / 2
    # Nusselt numbers
    Nu_bot[Ny] = []
    Nu_top[Ny] = []
    T_bulk_denom = 0
    for j in range(u.shape[1]):
        T_bulk_denom += cp * dy * u[0, j] * rho
    for i in range(T[Ny].shape[0]):
        T_bulk = 0
        for j in range(1, T[Ny].shape[1] - 1):
            if i * dx <= b and j * dy >= Ly / 2:
                continue
            T_bulk += rho * uT[i, j] * cp * T[Ny][i, j] * dy
        T_bulk /= T_bulk_denom
        T_bot, T_top = T[Ny][i, 0], T[Ny][i, Ny + 1]
        Nu_bot[Ny].append(2 * Ly * q / (k * (T_bot - T_bulk)))
        if i > b / dx:
            Nu_top[Ny].append(2 * Ly * q / (k * (T_top - T_bulk)))
    # Reattachment lengths
    u_top = uT[:, Ny]
    for i in range(int(np.floor(b / dx)) + 1, Nx):
        if u_top[i] > 0:
            m = (u_top[i] - u_top[i - 1]) / dx
            xr[Ny] = i * dx - (u_top[i] / m) - b
            break

print('Problem 1 reattachment (160xNy grid, Re = 200):')
for Ny, val in xr.items():
    exp = x_r(200)
    err = (val - exp) / exp * 100
    print('  Ny = {}, x_r = {:.4f}, exp = {:.4f}, err = {:.2f}%'.format(Ny, val, exp, err))

# Problem 1a
dy = Ly / 90
T_y = np.hstack((0, (np.linspace(dy / 2, Ly - dy / 2, num = T[90].shape[1] - 2)), Ly))
fig, ax = plt.subplots(1)
fig.set_figwidth(8)
fig.set_figheight(3)
```

```
for x in (Ly * np.array([6, 12, 24]) + b):
    i_mid = x / dx + 0.5
    i_min, i_max = int(np.floor(i_mid)), int(np.ceil(i_mid))
    ax.plot(T_y, (T[90][i_min, :] + T[90][i_max, :]) / 2, label='$x$ = {:.2f} m'.format(x), linewidth=1)
plt.xlabel('$y$ (m)')
plt.ylabel('$T(x, y)$')
plt.legend()
plt.grid()
plt.tight_layout()
fig.savefig('results/1a_Ty.pdf')

# Problem 1b
T_x = np.hstack((0, (np.linspace(dx / 2, Lx - dx / 2, num = T[90].shape[0] - 2)), Lx))
fig, ax = plt.subplots(1)
fig.set_figwidth(8)
fig.set_figheight(3)
after_step = np.where(T_x >= b)[0][0]
ax.plot(T_x, T[90][:, 0], label='Bottom wall', linewidth=1)
ax.plot(T_x[after_step:], T[90][after_step:, -1], label='Top wall'.format(Ly), linewidth=1)
plt.xlabel('$x$ (m)')
plt.ylabel('$T(x)$')
plt.legend()
plt.grid()
plt.tight_layout()
fig.savefig('results/1b_Twall.pdf')

# Problem 1b final values
print('Problem 1b bottom wall final: {:.3f}'.format(T[90][-1, 0]))
print('Problem 1b top wall final: {:.3f}'.format(T[90][-1, -1]))

# Problem 1c
fig, ax = plt.subplots(1, 2)
fig.set_figwidth(8)
fig.set_figheight(3)
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
i = 0
for Ny in Ny_vals:
    ax[0].plot(T_x[np.where(T_x > b)], Nu_top[Ny], label='$N_y = {}$'.format(Ny), linewidth=1)
    ax[1].plot(T_x, Nu_bot[Ny], linewidth=1)
ax[0].set_xlabel('$x$ (m)')
ax[1].set_xlabel('$x$ (m)')
ax[0].set_ylabel('Nu$(x)$')
ax[0].set_title('Top wall', fontsize=10)
ax[1].set_title('Bottom wall', fontsize=10)
ax[0].grid()
ax[1].grid()
handles, labels = ax[0].get_legend_handles_labels()
lgd = ax[0].legend(handles, labels, loc='lower center', bbox_to_anchor=(1.0, -0.37),
                ncol=4, fontsize=9)
fig.tight_layout()
fig.savefig('results/1c_Nu.pdf', bbox_inches='tight', bbox_extra_artists=(lgd,))

# Problem 1c final values
print('Problem 1c bottom Nu(Ny = 90) final: {:.3f}'.format(Nu_bot[90][-1]))
print('Problem 1c top Nu(Ny = 90) final: {:.3f}'.format(Nu_top[90][-1]))

################################################################################
# Problem 2

Ny = 70
dx = Lx / 160
dy = Ly / Ny

x_vals = (Ly * np.array([6, 12, 24]) + b)
Re_vals = [100, 300, 400]
xr = {}
u_cut, v_cut = {}, {}
for Re in Re_vals:
    u = np.loadtxt('results/Re{}_Nx160_Ny70_u.csv'.format(Re), delimiter=',').T
    v = np.loadtxt('results/Re{}_Nx160_Ny70_v.csv'.format(Re), delimiter=',').T
    T = np.loadtxt('results/Re{}_Nx160_Ny70_T.csv'.format(Re), delimiter=',').T
    # Cut along various xcome
    u_cut[Re], v_cut[Re] = {}, {}
    for x in x_vals:
        u_i = x / dx
        v_i = x / dx + 0.5
        u_i_min, u_i_max = int(np.ceil(u_i)), int(np.ceil(u_i))
        v_i_min, v_i_max = int(np.ceil(v_i)), int(np.ceil(v_i))
        u_cut[Re][x] = (u[u_i_min, :] + u[u_i_min, :]) / 2
        v_cut[Re][x] = (v[u_i_min, :] + v[u_i_min, :]) / 2
    # u at T points
```

```python
    uT = np.zeros(T.shape)
    for i in range(1, T.shape[0] - 1):
        for j in range(u.shape[1]):
            uT[0, j] = u[0, j]
            uT[u.shape[0], j] = u[u.shape[0] - 1, j]
            uT[i, j] = (u[i - 1, j] + u[i, j]) / 2
    # Reattachment length
    u_top = uT[:, Ny]
    for i in range(int(np.floor(b / dx)) + 1, Nx):
        if u_top[i] > 0:
            m = (u_top[i] - u_top[i - 1]) / dx
            xr[Re] = i * dx - (u_top[i] / m) - b
            break
print('Problem 2 reattachment (160x70 grid):')
for Re, val in xr.items():
    exp = x_r(Re)
    err = (val - exp) / exp * 100
    print('  Re = {}, x_r = {:.4f}, exp = {:.4f}, err = {:.2f}%'.format(Re, val, exp, err))

u_y = np.hstack((0, (np.linspace(dy / 2, Ly - dy / 2, num = len(u_cut[Re][x]) - 2)), Ly))
v_y = np.linspace(0, Ly, num=len(v_cut[Re][x]))

# u velocity profile
fig, ax = plt.subplots(1, 3)
fig.set_figwidth(8)
fig.set_figheight(3)
i = 0
for Re in Re_vals:
    for x in x_vals:
        ax[i].plot(u_y, u_cut[Re][x], label='x = {:.2f} m'.format(x), linewidth=1)
    ax[i].set_title('Re = {}'.format(Re), fontsize=10)
    ax[i].set_xlabel('$y$ (m)')
    ax[i].grid()
    i += 1
ax[0].set_ylabel('$u(x, y)$')
handles, labels = ax[1].get_legend_handles_labels()
lgd = ax[1].legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.37),
                ncol=3, fontsize=9)
fig.tight_layout()
fig.savefig('results/2_u.pdf', bbox_inches='tight', bbox_extra_artists=(lgd,))

# u velocity profile
fig, ax = plt.subplots(1, 3)
fig.set_figwidth(8)
fig.set_figheight(3)
i = 0
for Re in Re_vals:
    for x in x_vals:
        ax[i].plot(v_y, v_cut[Re][x], label='x = {:.2f} m'.format(x), linewidth=1)
    ax[i].set_title('Re = {}'.format(Re), fontsize=10)
    ax[i].set_xlabel('$y$ (m)')
    ax[i].grid()
    i += 1
ax[0].set_ylabel('$v(x, y)$')
handles, labels = ax[1].get_legend_handles_labels()
lgd = ax[1].legend(handles, labels, loc='lower center', bbox_to_anchor=(0.5, -0.37),
                ncol=3, fontsize=9)
fig.tight_layout()
fig.savefig('results/2_v.pdf', bbox_inches='tight', bbox_extra_artists=(lgd,))
```