MEEN 644 - Homework 3

Logan Harbour
February 17, 2019

Problem statement

Consider a thin copper square plate of dimensions 0.5 m $\times$ 0.5 m. The temperature of the west and south edges are maintained at 50 °C and the north edge is maintained at 100 °C. The east edge is insulated. Using finite volume method, write a program to predict the steady-state temperature solution.

(a) **(35 points)** Set the over relaxation factor α from 1.00 to 1.40 in steps of 0.05 to identify α_{opt}. Plot the number of iterations required for convergence for each α.

(b) **(15 points)** Solve the same problem using $21^2, 25^2, 31^2$, and 41^2 CVs, respectively. Plot the temperature at the center of the plate (0.25 m, 0.25 m) vs CVs.

(c) **(10 points)** Plot the steady state temperature contour in the 2D domain with the 41^2 CV solution.

Preliminaries

Two-dimensional heat conduction

With two-dimensional heat conduction with constant material properties, insulation on the right and prescribed temperatures on all other sides, we have the PDE

$$\begin{cases} k\frac{\partial^2 T}{\partial x^2} + k\frac{\partial^2 T}{\partial y^2} = 0\,, \\ T(x,0) = T_B\,, \\ T(0,y) = T_L\,, \\ T(0,L) = T_T\,, \\ -k\frac{\partial T}{\partial x}\Big|_{x=L} = 0\,, \end{cases} \tag{1}$$

where

$$T_B \equiv 50\ ^\circ\text{C}\,, \qquad T_L \equiv 50\ ^\circ\text{C}\,, \qquad T_T \equiv 100\ ^\circ\text{C}\,.$$
$$k \equiv 386\ \text{W/m}\ ^\circ\text{C}\,, \qquad L \equiv 0.5\ \text{m}\,.$$

We discretize the region on $x \times y = [0, L]^2$ by N^2 internal nodes with $\Delta x = x/N, \Delta y = y/N$.

Control volume equations

Integrate over an internal control volume (i, j) to obtain

$$k \iint_{CV_{i,j}} \left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] dxdy = 0 \,, \quad (i,j) \in [2, 3, \ldots, N-1]^2 \,.$$

$$k\Delta y \left[\frac{\partial T}{\partial x}\Big|_{w_{i,j}} - \frac{\partial T}{\partial x}\Big|_{e_{i,j}} \right] + k\Delta x \left[\frac{\partial T}{\partial y}\Big|_{n_{i,j}} - \frac{\partial T}{\partial y}\Big|_{s_{i,j}} \right] = 0 \,, \quad (i,j) \in [2, 3, \ldots, N-1]^2 \,.$$

Now use the two node formulation for the derivative terms to obtain

$$k\Delta y \left[\frac{T_{E_{ij}} - T_{P_{ij}}}{\Delta x} - \frac{T_{P_{ij}} - T_{W_{ij}}}{\Delta x} \right] + k\Delta x \left[\frac{T_{N_{ij}} - T_{P_{ij}}}{\Delta y} - \frac{T_{P_{ij}} - T_{S_{ij}}}{\Delta y} \right] = 0 \,, \quad (i,j) \in [2, 3, \ldots, N-1]^2 \,.$$

Collect like terms and modify the index to obtain

$$T_{i,j}a_p - T_{i,j+1}a_n - T_{i+1,j}a_e - T_{i,j-1}a_s - T_{i-1,j}a_w = 0 \,, \quad (i,j) \in [2, 3, \ldots, N-1]^2 \,, \tag{2}$$

where (note that the below applies only to *internal* control volumes)

$$a_n \equiv \frac{k\Delta y}{\Delta x} \,, \quad a_e \equiv \frac{k\Delta x}{\Delta y} \,, \quad a_s \equiv \frac{k\Delta y}{\Delta x} \,, \quad a_w \equiv \frac{k\Delta x}{\Delta y} \,, \quad a_p \equiv a_n + a_e + a_s + a_w \,.$$

The remaining equations are solved similarly but with slight differences depending on which boundary the CV is on; for example, for control volumes on the left boundary we instead have $a_w = 2k\Delta y/\Delta x$.

Solving method

The problem is to be solved by the line-by-line method. In specific, the sweeping arrangement is: **south to north, west to east, north to south, east to west**. In this method, the contribution from one direction in a given control volume is lagged and moved to the right hand side in order to solve a tri-diagonal system. Convergence is declared when

$$R = \sum_{\text{CV}} \left| a_p T_p - \sum_{\text{nb}} a_{\text{nb}} T_{\text{nb}} b_p \right| \leq 10^{-5} \,. \tag{3}$$

Upon solving an individual system $Ax^{\ell+1} = b$ with relaxation (where ℓ is the iteration index), the system is relaxed with the coefficient α by modifying it after construction by

$$\begin{cases} a_{ii} = a_{ii}/\alpha \,, \\ b_i = b_i + (\alpha^{-1} - 1)a_{ii}x_i^{\ell} \,, \end{cases} \qquad i = 1, \ldots, N \,,$$

and it is then solved using the standard TDMA algorithm.

Results

Part a

With the given range of α, it was determined for this specific problem with 15^2 CVs that $\alpha_{\text{opt}} \approx 1.3$ (given that it required the fewest iterations). The solve did not converge (with an attempted 1,000 iterations) for $\alpha = 1.4$. The requested figure showing the residuals follows in Figure 1.

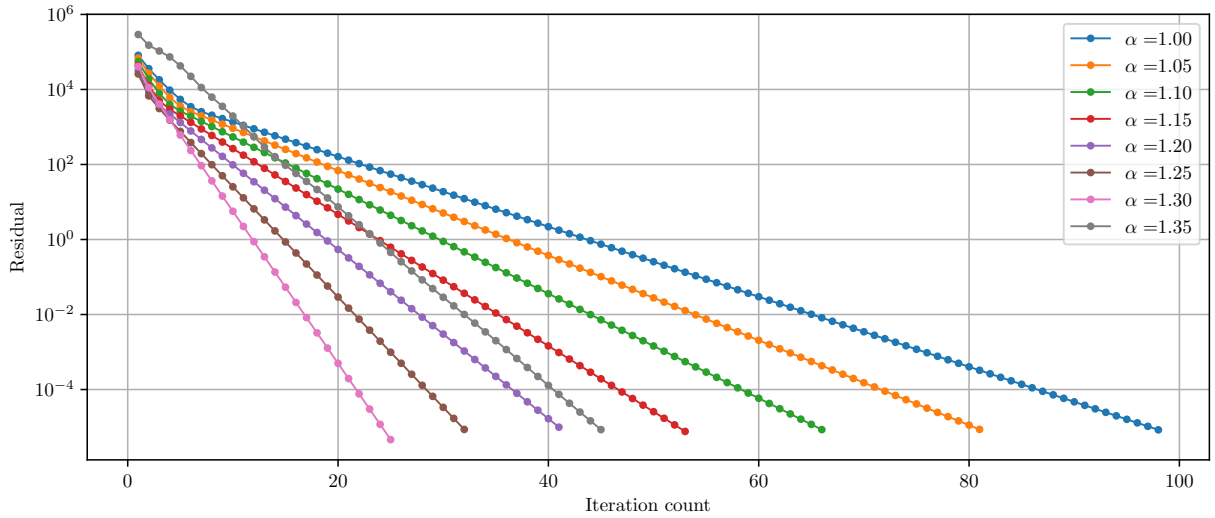

Figure 1: Plot of the residual vs. iteration count for each relaxation parameter, α.

Part b

With a mesh refinement of $21^2, 25^2, 31^2$, and 41^2 CVs, the center temperature for each refinement with varying relaxation parameter α is plotted below in Figure 2. The same result is tabulated in Table **??**.

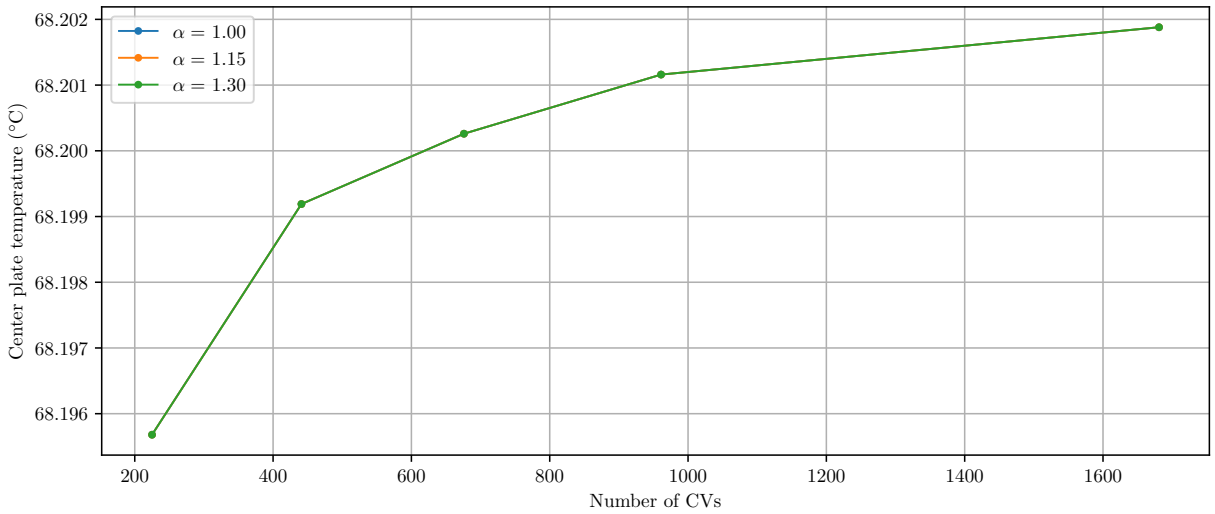

Figure 2: Plot of the center temperature with mesh refinement.

Table 1: The temperature at the center of the plate with varying refinements and relaxation parameters.

CVs	$\alpha = 1.0$	$\alpha = 1.15$	$\alpha = 1.30$
225	68.19568	68.19568	68.19568
441	68.19919	68.19919	68.19919
625	68.20026	68.20026	68.20026
961	68.20116	68.20116	68.20116
1681	68.20188	68.20188	68.20188

With refinement, the iteration requirement was plotted and is seen below in Figure 3. No refinement levels converged for $\alpha = 1.4$ and only the 225 CV case converged for $\alpha = 1.35$.

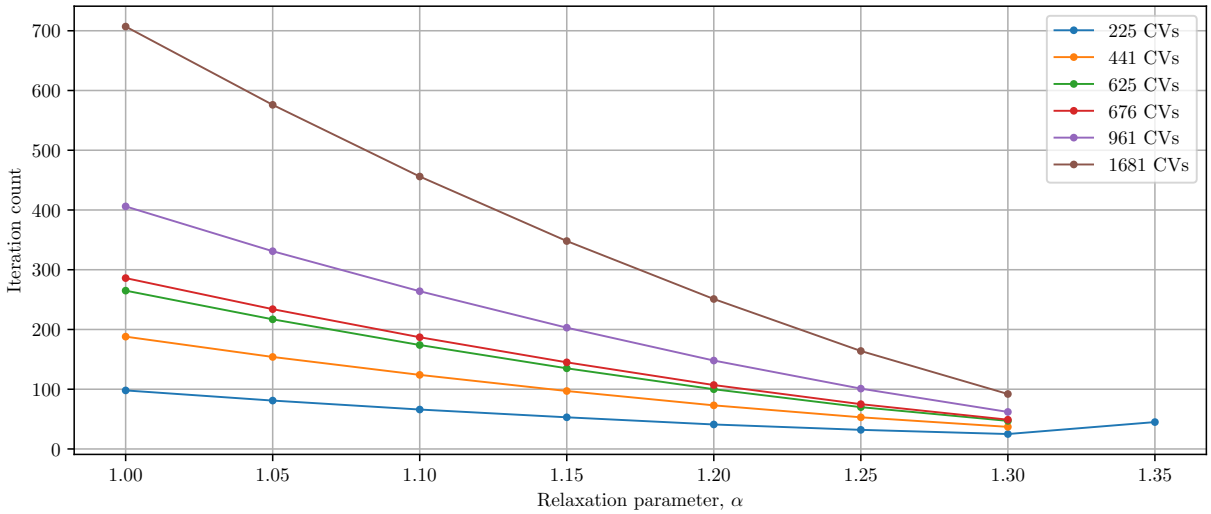

Figure 3: Plot of the required iterations with mesh refinement.

Part c

With the final mesh refinement of 41^2 CVs, a colored contour plot of the temperature solution follows in Figure 4.

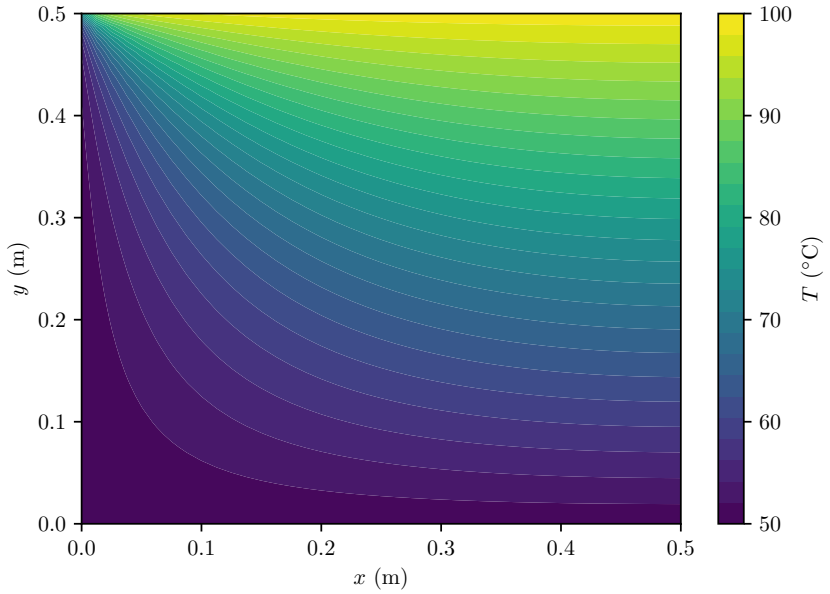

Figure 4: Plot of the solution with 41^2 CVs.

Code listing

For the implementation, we have the following files:

- `Makefile` – Allows for compiling the c++ project with `make`.
- `hwk3.cpp` – Contains the `main()` function that is required by C that runs the cases requested in this problem set.
- `Conduction2D.h` / `Conduction2D.cpp` – Contains the `Conduction2D` class which is the solver for the 2D heat conduction problem required in this homework.
- `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()`.
- `plots.py` - Produces the plots in this report.

Makefile

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

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

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

hwk3.cpp

```cpp
#include "Conduction2D.h"
#include <map>

template <typename T> void save(const std::vector<T> &v, std::string filename) {
  std::ofstream f;
  f.open(filename);
  for (unsigned int i = 0; i < v.size(); ++i)
    f << std::scientific << v[i] << std::endl;
  f.close();
}

int main() {
  // Part a: change relaxation factor
  std::cout << "Part a" << std::endl;
  for (double w : {1.0, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.35, 1.4}) {
    std::cout << "  Running with w = " << w << ": ";
    Conduction2D problem(15, 15, w);
    problem.solve();
    save(problem.getResiduals(), "results/a/" + std::to_string(w) + ".csv");
  }
  std::cout << std::endl;

  // Part b: refine mesh and retreive center value
  std::cout << "Part b" << std::endl;
  std::map<double, std::vector<double>> temps;
  for (unsigned int N : {15, 21, 25, 31, 41}) {
    std::cout << "  Running with " << N * N << " CVs: " << std::endl;

    std::vector<unsigned int> iterations;

    for (double w : {1.0, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.35}) {
      std::cout << "    Running with w = " << w << ": ";

      Conduction2D problem(N, N, w);
      problem.solve();

      // Store iteration count and center solution if it converged
      if (problem.getNumIterations() != 1000) {
        iterations.push_back(problem.getNumIterations());
        temps[w].push_back(problem.getT((N - 1) / 2, (N - 1) / 2));
      }

      // Part c: save solution for 41x41
      if (N == 41 && w == 1.0)
        problem.saveT("results/c/solution.csv");
    }

    // Save iteration count and center solution
    save(iterations, "results/b-iterations/" + std::to_string(N) + ".csv");
  }
```

```
  for (double w : {1.0, 1.15, 1.3})
    save(temps[w], "results/b-temps/" + std::to_string(w) + ".csv");
  return 0;
}
```

Conduction2D.h

```
#ifndef CONDUCTION2D_H
#define CONDUCTION2D_H

#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>

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

/**
 * Solves a 2D heat conduction problem with dirichlet conditions on the top,
 * left, bottom and with a zero-flux condition on the right with Nx x Ny
 * internal control volumes.
 */
class Conduction2D {
public:
  Conduction2D(unsigned int Nx, unsigned int Ny, double alpha, double Lx = 0.5,
               double Ly = 0.5, double k = 386.0, double T_L = 50.0,
               double T_T = 100.0, double T_B = 50.0, double tol = 1e-5,
               unsigned int max_its = 1000);

  void solve();

  // Get the solution at the (i, j) internal node
  const double getT(unsigned int i, unsigned int j) const { return T(i, j); }

  // Get the residuals and number of iterations
  const std::vector<double> &getResiduals() const { return residuals; }
  unsigned int getNumIterations() { return residuals.size(); }

  // Save the solution
  void saveT(std::string filename) const { T.save(filename); }

private:
  double computeResidual() const;

  // Precompute operations
  void precomputeProperties();
  void precomputeColumn(unsigned int col);
  void precomputeRow(unsigned int row);

  // Solve and sweep operations
  void solveColumn(unsigned int col);
  void solveRow(unsigned int row);

protected:
  // Number of interior nodal points in the x and y-dimensions
  const unsigned int Nx, Ny;
```

```
  // Geometry [m]
  const double Lx, Ly, dx, dy;
  // Heat conduction coefficient [W / m k]
  const double k;
  // Dirichlet oundary conditions (left, top, bottom) [C]
  const double T_L, T_T, T_B;
  // Properties stored in matrix form
  Matrix a_p, a_n, a_e, a_s, a_w;

  // Inverse of the relaxation coefficient
  const double w_inv;
  // Iteration tolerance
  const double tol;
  // Maximum iterations
  const unsigned int max_its;

  // Temperature solution
  Matrix T;

  // Precomputed matrices for the TDMA solves
  std::vector<TriDiagonal> pre_A_x, pre_A_y;
  // Precomputed RHS for the TDMA solves
  std::vector<std::vector<double>> pre_b_x, pre_b_y;
  // Matrices for the TDMA solves
  TriDiagonal A_x, A_y;
  // RHS/solution vector for the TDMA solves
  std::vector<double> b_x, b_y;
  // Residual for each iteration
  std::vector<double> residuals;
};

#endif /* CONDUCTION2D_H */
```

Conduction2D.cpp

```
#include "Conduction2D.h"

Conduction2D::Conduction2D(unsigned int Nx, unsigned int Ny, double w,
                           double Lx, double Ly, double k, double T_L,
                           double T_T, double T_B, double tol,
                           unsigned int max_its)
    : // Interior nodal points
      Nx(Nx), Ny(Ny), Lx(Lx), Ly(Ly), dx(Lx / Nx), dy(Ly / Ny),
      // Material properties
      k(k),
      // Boundary conditions
      T_L(T_L), T_T(T_T), T_B(T_B),
      // Material properties in matrix form
      a_p(Nx, Ny), a_n(Nx, Ny), a_e(Nx, Ny), a_s(Nx, Ny), a_w(Nx, Ny),
      // Solver properties
      w_inv(1.0 / w), tol(tol), max_its(max_its),
      // Initialize matrices and vectors
      T(Nx, Ny, (T_L + T_T + T_B) / 3.0), pre_A_x(Ny, Nx), pre_A_y(Ny, Nx),
      pre_b_x(Ny, std::vector<double>(Nx)),
      pre_b_y(Nx, std::vector<double>(Ny)), A_x(Nx), A_y(Ny), b_x(Nx), b_y(Ny) {
}
```

```
void Conduction2D::solve() {
  // Compute the a coefficients for each CV
  precomputeProperties();
  // Compute the unchanging LHS and RHS for each column
  for (unsigned int i = 0; i < Nx; ++i)
    precomputeColumn(i);
  // Compute the unchanging LHS and RHS for each row
  for (unsigned int j = 0; j < Ny; ++j)
    precomputeRow(j);

  // Iterate and exit when complete
  for (unsigned int l = 1; l <= max_its; ++l) {
    // Sweep south to north
    for (int j = 0; j < Ny; ++j)
      solveRow(j);
    // Sweep west to east
    for (int i = 0; i < Nx; ++i)
      solveColumn(i);
    // Sweep north to south
    for (int j = Ny - 1; j >= 0; --j)
      solveRow(j);
    // Sweep east to west
    for (int i = Nx - 1; i >= 0; --i)
      solveColumn(i);

    // Check for convergence and store residual
    double R = computeResidual();
    residuals.push_back(R);
    if (R < tol) {
      std::cout << "Converged with " << l << " iterations" << std::endl;
      return;
    }
  }

  std::cout << "Failed to converge in " << max_its << " iterations"
            << std::endl;
}

void Conduction2D::precomputeProperties() {
  // Set all neighbors to the default at first
  a_n = k * dx / dy;
  a_e = k * dy / dx;
  a_s = k * dx / dy;
  a_w = k * dy / dx;

  // Top Dirichlet
  a_n.setRow(Ny - 1, 2 * k * dx / dy);
  // Right Neumann
  a_e.setColumn(Nx - 1, 0);
  // Bottom Dirichlet
  a_s.setRow(0, 2 * k * dx / dy);
  // Left dirichlet
  a_w.setColumn(0, 2 * k * dy / dx);

  // Center point
  for (unsigned int i = 0; i < Nx; ++i)
    for (unsigned int j = 0; j < Ny; ++j)
      a_p(i, j) = a_n(i, j) + a_e(i, j) + a_s(i, j) + a_w(i, j);
```

```
}

void Conduction2D::precomputeRow(unsigned int j) {
  TriDiagonal &A = pre_A_x[j];
  std::vector<double> &b = pre_b_x[j];

  // First treat all as an internal volume
  A.addTopRow(a_p(0, j) * w_inv, -a_e(0, j));
  A.addBottomRow(-a_w(Nx - 1, j), a_p(Nx - 1, j) * w_inv);
  for (unsigned int i = 1; i < Nx - 1; ++i)
    A.addMiddleRow(i, -a_w(i, j), a_p(i, j) * w_inv, -a_e(i, j));

  // Left Dirichlet
  b[0] += T_L * a_w(0, j);
  // Top dirichlet
  if (j == Ny - 1)
    for (unsigned int i = 0; i < Nx; ++i)
      b[i] += T_T * a_n(i, j);
  // Bottom dirichlet
  else if (j == 0)
    for (unsigned int i = 0; i < Nx; ++i)
      b[i] += T_B * a_s(i, j);
}

void Conduction2D::precomputeColumn(unsigned int i) {
  TriDiagonal &A = pre_A_y[i];
  std::vector<double> &b = pre_b_y[i];

  // First treat all as an internal volume
  A.addTopRow(a_p(i, 0) * w_inv, -a_n(i, 0));
  A.addBottomRow(-a_s(i, Ny - 1), a_p(i, Ny - 1) * w_inv);
  for (unsigned int j = 1; j < Ny - 1; ++j)
    A.addMiddleRow(j, -a_s(i, j), a_p(i, j) * w_inv, -a_n(i, j));

  // Left Dirichlet
  if (i == 0)
    for (unsigned int j = 0; j < Ny; ++j)
      b[j] += T_L * a_w(i, j);
  // Top Dirichlet
  b[Ny - 1] += T_T * a_n(i, Ny - 1);
  // Bottom Dirichlet
  b[0] += T_B * a_s(i, 0);
}

void Conduction2D::solveRow(unsigned int j) {
  // Copy pre-filled Ax = b for this row
  A_x = pre_A_x[j];
  b_x = pre_b_x[j];

  // RHS contribution from volumes above and below
  if (j > 0)
    for (unsigned int i = 0; i < Nx; ++i)
      b_x[i] += T(i, j - 1) * a_s(i, j);
  if (j < Ny - 1)
    for (unsigned int i = 0; i < Nx; ++i)
      b_x[i] += T(i, j + 1) * a_n(i, j);

  // Relax, solve, and store solution (which is in b_x)
  if (w_inv != 1)
```

```
    for (unsigned int i = 0; i < Nx; ++i)
      b_x[i] += (w_inv - 1.0) * a_p(i, j) * T(i, j);
  A_x.solveTDMA(b_x);
  T.setRow(j, b_x);
}

void Conduction2D::solveColumn(unsigned int i) {
  // Copy pre-filled Ax = b for this row
  A_y = pre_A_y[i];
  b_y = pre_b_y[i];

  // RHS contribution from volumes left and right
  if (i > 0)
    for (unsigned int j = 0; j < Ny; ++j)
      b_y[j] += T(i - 1, j) * a_w(i, j);
  if (i < Nx - 1)
    for (unsigned int j = 0; j < Ny; ++j)
      b_y[j] += T(i + 1, j) * a_e(i, j);

  // Relax, solve, and store solution (which is in b_y)
  if (w_inv != 1)
    for (unsigned int j = 0; j < Ny; ++j)
      b_y[j] += (w_inv - 1.0) * a_p(i, j) * T(i, j);
  A_y.solveTDMA(b_y);
  T.setColumn(i, b_y);
}

double Conduction2D::computeResidual() const {
  double R = 0.0, val = 0.0;
  // Sum over all CVs
  for (unsigned int i = 0; i < Nx; ++i)
    for (unsigned int j = 0; j < Ny; ++j) {
      // Main diagonal contribution and pre-computed RHS (from BC)
      val = a_p(i, j) * T(i, j) - pre_b_y[i][j];
      // Not on left boundary
      if (i > 0)
        val -= a_w(i, j) * T(i - 1, j);
      // Not on right boundary
      if (i < Nx - 1)
        val -= a_e(i, j) * T(i + 1, j);
      // Not on bottom boundary
      if (j > 0)
        val -= a_s(i, j) * T(i, j - 1);
      // Not top boundary
      if (j < Ny - 1)
        val -= a_n(i, j) * T(i, j + 1);
      R += std::abs(val);
    }
  return R;
}
```

Matrix.h

```
#ifndef MATRIX
#define MATRIX

#define NDEBUG
```

```cpp
#include <cassert>
#include <vector>

/**
 * Class that holds a Nx x Ny matrix with common matrix operations.
 */
class Matrix {
public:
  Matrix(unsigned int Nx, unsigned int Ny, double v = 0)
      : Nx(Nx), Ny(Ny), M(Nx, std::vector<double>(Ny, v)) {}

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

  // Saves the matrix in csv format
  void save(const std::string filename, unsigned int precision = 12) const {
    std::ofstream f;
    f.open(filename);
    for (unsigned int j = 0; j < Ny; ++j) {
      for (unsigned int i = 0; i < Nx; ++i) {
        if (i > 0)
          f << ",";
        f << std::setprecision(precision) << M[i][j];
      }
      f << std::endl;
    }
    f.close();
  }

  // Set the j-th row to v
  void setRow(unsigned int j, double v) {
    assert(j < Ny);
    for (unsigned int i = 0; i < Nx; ++i)
      M[i][j] = v;
  }
  // Set the i-th column to v
  void setColumn(unsigned int i, double v) {
    assert(i < Nx);
    for (unsigned int j = 0; j < Ny; ++j)
      M[i][j] = v;
  }

  // Set the j-th row to vs
  void setRow(unsigned int j, std::vector<double> &vs) {
    assert(j < Ny && vs.size() == Nx);
    for (unsigned int i = 0; i < Nx; ++i)
```

```
      M[i][j] = vs[i];
  }
  // Set the i-th column to vs
  void setColumn(unsigned int i, std::vector<double> &vs) {
    assert(i < Nx && vs.size() == Ny);
    for (unsigned int j = 0; j < Ny; ++j)
      M[i][j] = vs[j];
  }

private:
  // The size of this matrix
  const unsigned int Nx, Ny;

  // Matrix storage
  std::vector<std::vector<double> > M;
};

#endif /* MATRIX_H */
```

TriDiagonal.h

```
#ifndef TRIDIAGONAL_H
#define TRIDIAGONAL_H

#define NDEBUG
#include <cassert>

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

  // Operator for setting the entire matrix to a value
  void operator=(TriDiagonal & from) {
    assert(from.getN() == N);
    A = from.getA();
    B = from.getB();
    C = from.getC();
  }

  // Gets the value of the (i, j) entry
  const double operator()(unsigned int i, unsigned int j) const {
    assert(i < N && j > i - 2 && j < i + 2);
    if (j == i - 1)
      return A[i];
    else if (j == i)
      return B[i];
    else if (j == i + 1)
      return C[i];
    else {
      std::cerr << "( " << i << "," << j << ") out of TriDiagonal system";
      std::terminate();
    }
```

```
  }

  // Adders for the top, middle, and bottom rows
  void addTopRow(double b, double c) {
    B[0] += b;
    C[0] += c;
  }
  void addMiddleRow(unsigned int i, double a, double b, double c) {
    assert(i < N - 1 && i != 0);
    A[i] += a;
    B[i] += b;
    C[i] += c;
  }
  void addBottomRow(double a, double b) {
    A[N - 1] += a;
    B[N - 1] += b;
  }

  // Getters for the raw vectors
  const std::vector<double> &getA() const { return A; }
  const std::vector<double> &getB() const { return B; }
  const std::vector<double> &getC() const { return C; }

  // Getter for the size
  unsigned int getN() { return N; }

  // Solves the system Ax = d in place where d eventually stores the solution
  void solveTDMA(std::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 != std::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;

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

#endif /* TRIDIAGONAL_H */
```

plots.py

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

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

# Problem a
fig = plt.figure()
fig.set_figwidth(9)
fig.set_figheight(4)
for file in glob.glob('results/a/*.csv'):
    alpha = float(os.path.splitext(file)[0].split("/")[-1])
    if alpha == 1.4:
        continue
    residuals = np.loadtxt(file)
    iterations = list(range(1, len(residuals) + 1))
    plt.semilogy(iterations, residuals, '.-', linewidth=1,
                 label=r'$\alpha = ${:.2f}'.format(alpha))
plt.xlabel(r'Iteration count')
plt.ylabel(r'Residual')
plt.legend()
plt.tight_layout()
plt.grid()
fig.savefig('results/a.pdf')

# Problem b number of iterations
fig = plt.figure()
fig.set_figwidth(9)
fig.set_figheight(4)
alphas = [1.0, 1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.35]
for file in glob.glob('results/b-iterations/*.csv'):
    N = int(os.path.splitext(file)[0].split("/")[-1])
    iterations = np.loadtxt(file)
    plt.plot(alphas[0:len(iterations)], iterations, '.-', linewidth=1,
             label=r'{} CVs'.format(N * N))
plt.xlabel(r'Relaxation parameter, $\alpha$')
plt.ylabel(r'Iteration count')
plt.legend()
plt.tight_layout()
plt.grid()
fig.savefig('results/b-iterations.pdf')

# Problem b temps
fig = plt.figure()
fig.set_figwidth(9)
fig.set_figheight(4)
Ns = [15**2, 21**2, 26**2, 31**2, 41**2]
for file in glob.glob('results/b-temps/*.csv'):
    alpha = float(os.path.splitext(file)[0].split("/")[-1])
    temps = np.loadtxt(file)
    plt.plot(Ns, temps, '.-', linewidth=1, label=r'$\alpha =$ {:.2f}'.format(alpha))
plt.xlabel(r'Number of CVs')
plt.ylabel(r'Center plate temperature ($^\circ$C)')
plt.ticklabel_format(useOffset=False)
```

```
plt.legend()
plt.tight_layout()
plt.grid()
fig.savefig('results/b-temps.pdf')

# Problem c
T = np.loadtxt('results/c/solution.csv', delimiter=',')
fig = plt.figure()
fig.set_figwidth(5.5)
fig.set_figheight(4)
x, y = np.meshgrid(np.linspace(0, 0.5, num=41), np.linspace(0, 0.5, num=41))
plot = plt.contourf(x, y, T, levels=np.linspace(50, 100, num=25))
cbar = fig.colorbar(plot)
cbar.set_ticks(np.arange(50, 101, 10))
cbar.ax.set_ylabel(r'$T$ ($^\circ$C)')
plt.xlabel('$x$ (m)')
plt.ylabel('$y$ (m)')
plt.tight_layout()
fig.savefig('results/c.pdf')
```