Annex: Code of 2D resolution of 4 conductive materials

Authors: Gerard Morales Riera
Gas dynamics and heat transfer, Carlos David Pérez Segarra

Aerospace Technologies Engineering degree, ESEIAAT

Main code (*C*++)

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <chrono>
#include <algorithm>
#include "json.hpp"

using namespace std;
using json = nlohmann::json;

// Variables ------------------------------------------------------------
// Define all the variables, including the j object to read the json file
json j;
int i, t, N, M, max_iter, wrong_time_iter;
double T0, mytime, V, VP, dx, dy, delta, rf, dt, ti, tf, nt, L, H, W, TwBottom, TwRight_0, TwRight_slope, qwTop, Text, alpha_ext, Se, Sw, Sn, Ss, SwallH, SwallL;
vector<double> p1, p2, p3, rho, cp, lambda, xP, yP;
vector<vector<int>> mat;
vector<vector<double>> aP, aE, aW, aN, aS, bP;
vector<vector<vector<double>>> T;

// Functions ------------------------------------------------------------
// Overload the - operator for matrices
vector<vector<double>> operator-(const vector<vector<double>>& a, const vector<vector<double>>& b) {
    // Create the result matrix
    vector<vector<double>> result = a;

    // Subtract the matrices
    for(size_t i = 0; i < a.size(); i++) {
        for(size_t j = 0; j < a[i].size(); j++) {
            result[i][j] -= b[i][j];
        }
    }

    return result;
}

// Overload the + operator for matrices
vector<vector<double>> operator+(const vector<vector<double>>& a, const vector<vector<double>>& b) {
    // Create the result matrix
    vector<vector<double>> result = a;

    // Add the matrices
    for(size_t i = 0; i < a.size(); i++) {
        for(size_t j = 0; j < a[i].size(); j++) {
            result[i][j] += b[i][j];
```

```
        }
    }

    return result;
}

// Overload the * operator for matrices times scalars
vector<vector<double>> operator*(const vector<vector<double>>& a, const double& b) {
    // Create the result matrix
    vector<vector<double>> result = a;

    // Multiply the matrix by the scalar
    for(size_t i = 0; i < a.size(); i++) {
        for(size_t j = 0; j < a[i].size(); j++) {
            result[i][j] *= b;
        }
    }

    return result;
}

// Absolute value of matrix elements stored in a vector
vector<double> abs_mat_to_vec(const vector<vector<double>>& a) {
    // Create the result matrix
    vector<double> result;

    // Calculate the absolute value of the matrix
    for(size_t i = 0; i < a.size(); i++) {
        for(size_t j = 0; j < a[i].size(); j++) {
            result.push_back(abs(a[i][j]));
        }
    }

    return result;
}

// Get the correct lambda
double harmonic_lambda(int a, int b, string position) {
    // Check the position and return the correct lambda
    if(position == "E") {
        return 2*lambda[mat[a][b]]*lambda[mat[a][b+1]]/(lambda[mat[a][b]] + lambda[mat[a][b+1]]);
    } else if(position == "W") {
        return 2*lambda[mat[a][b-1]]*lambda[mat[a][b]]/(lambda[mat[a][b-1]] + lambda[mat[a][b]]);
    } else if(position == "N") {
        return 2*lambda[mat[a][b]]*lambda[mat[a-1][b]]/(lambda[mat[a][b]] + lambda[mat[a-1][b]]);
    } else if(position == "S") {
        return 2*lambda[mat[a+1][b]]*lambda[mat[a][b]]/(lambda[mat[a+1][b]] + lambda[mat[a][b]]);
    } else {
        cout << "Invalid position!" << endl;
        exit(0);
        return 0.0;
    }
}

// Actions ---------------------------------------------------------------
// Assign the constants from the json file to the variables
void set_constants() {
    // Read the file
    ifstream json_file("src/data.json");
    json_file >> j;

    // Set the Geometry
```

```
    p1.resize(2, 0.0);
    p2.resize(2, 0.0);
    p3.resize(2, 0.0);
    p1 = j["Geometry"]["p1"].get<vector<double>>();
    p2 = j["Geometry"]["p2"].get<vector<double>>();
    p3 = j["Geometry"]["p3"].get<vector<double>>();

    // General dimensions
    L = p3[0];
    H = p3[1];

    // Set the Material Properties
    rho.resize(4, 0.0);
    cp.resize(4, 0.0);
    lambda.resize(4, 0.0);
    for(i = 0; i < 4; i++) {
        rho[i] = j["Material Properties"][i]["rho"];
        cp[i] = j["Material Properties"][i]["cp"];
        lambda[i] = j["Material Properties"][i]["lambda"];
    }

    // Set the Boundary Conditions
    TwBottom = j["Boundary Conditions"]["TwBottom"];
    TwRight_0 = j["Boundary Conditions"]["TwRight"][0];
    TwRight_slope = j["Boundary Conditions"]["TwRight"][1];
    qwTop = j["Boundary Conditions"]["qwTop"];
    Text = j["Boundary Conditions"]["Text"];
    alpha_ext = j["Boundary Conditions"]["alpha_ext"];

    // Set the Control Volumes
    N = j["Control Volumes"]["N"];
    M = j["Control Volumes"]["M"];
    dx = L/M;
    dy = H/N;

    // Set the Time Parameters
    dt = j["Time Parameters"]["dt"];
    ti = j["Time Parameters"]["ti"];
    tf = j["Time Parameters"]["tf"];
    nt = (tf-ti)/dt;

    // Set the Solver Parameters
    T0 = j["Solver Parameters"]["T0"];
    rf = j["Solver Parameters"]["Relaxation Factor"];
    max_iter = j["Solver Parameters"]["Maximum Iterations"];
    delta = j["Solver Parameters"]["delta"];
}

// Find the material properties for each node
void find_materials() {
    // Resize the matrix
    mat.resize(N+2, vector<int>(M+2, 0));

    // Find the materials
    for(int i = 0; i < N+2; i++) {
        for(int j = 0; j < M+2; j++) {
            if(xP[j] <= p1[0] && yP[i] <= p1[1]) {
                mat[i][j] = 0;
            } else if(xP[j] > p1[0] && yP[i] <= p2[1]) {
                mat[i][j] = 1;
            } else if(xP[j] <= p2[0]) {
                mat[i][j] = 2;
            } else {
                mat[i][j] = 3;
            }
        }
```

```
    }

    // Create the materials file
    ofstream mater;
    mater.open("output/materials.txt");

    // Print the materials
    for(int i = 0; i < N+2; i++) {
        for(int j = 0; j < M+2; j++) {
            mater << mat[i][j] << " ";
        }
        mater << endl;
    }
    mater.close();
}

// Set the discretization constants that do not change over time
void discretization_constants() {
    // Initialize values
    aP.resize(N+2, vector<double>(M+2, 0.0));
    aE.resize(N+2, vector<double>(M+2, 0.0));
    aW.resize(N+2, vector<double>(M+2, 0.0));
    aN.resize(N+2, vector<double>(M+2, 0.0));
    aS.resize(N+2, vector<double>(M+2, 0.0));
    bP.resize(N+2, vector<double>(M+2, 0.0));

    // i = 1; j = 2, ..., M+1 (TOP)
    for(int j = 1; j < M+1; j++) {
        aS[0][j] = harmonic_lambda(0, j, "S")/(abs(yP[0]-yP[1]))*Ss;
        aP[0][j] = aS[0][j];
        bP[0][j] = qwTop*Sn;
    }

    // i = 2, ..., N+1; j = 2, ..., M+1 (MIDDLE)
    for(int i = 1; i < N+1; i++) {
        for(int j = 1; j < M+1; j++) {
            aW[i][j] = harmonic_lambda(i, j, "W")/(abs(xP[j]-xP[j-1]))*Sw;
            aE[i][j] = harmonic_lambda(i, j, "E")/(abs(xP[j+1]-xP[j]))*Se;
            aN[i][j] = harmonic_lambda(i, j, "N")/(abs(yP[i]-yP[i-1]))*Sn;
            aS[i][j] = harmonic_lambda(i, j, "S")/(abs(yP[i+1]-yP[i]))*Ss;
            aP[i][j] = aE[i][j] + aW[i][j] + aN[i][j] + aS[i][j] + rho[mat[i][j]]*cp[mat[i
    ][j]]*VP/dt;
        }
    }

    // i = 2, ..., N+1; j = 1 (LEFT)
    for(int i = 1; i < N+1; i++) {
        aE[i][0] = harmonic_lambda(i, 0, "E")/(abs(xP[1]-xP[0]))*Se;
        aP[i][0] = aE[i][0] + alpha_ext*Sw;
        bP[i][0] = alpha_ext*Sw*Text;
    }
}

// Perform initial calculations
void previous_calculations() {
    // Geometry properties
    V = L*H;
    SwallH = H;
    SwallL = L;
    xP.resize(M+2, 0.0);
    yP.resize(N+2, 0.0);
    xP[0] = 0.0;
    yP[0] = H;
    xP[1] = dx/2;
    yP[1] = H - dy/2;
    xP[M+1] = L;
```

```
    yP[N+1] = 0.0;

    // Control volume x positions
    for(i = 2; i < M+1; i++) {
        xP[i] = xP[i-1] + dx;
    }

    // Control volume y positions
    for(i = 2; i < N+1; i++) {
        yP[i] = yP[i-1] - dy;
    }

    // Control volume properties (structured mesh)
    VP = dx*dy;
    Sn = dx;
    Ss = dx;
    Sw = dy;
    Se = dy;

    // Find material properties for each node
    find_materials();

    // Set the discretization constants that do not change over time
    discretization_constants();
}

// Solve the problem with Gauss-Seidel
void solve_with_GS() {
    // Create the T_unsolved matrix
    vector<vector<double>> T_unsolved;
    T_unsolved.resize(N+2, vector<double>(M+2, T0));

    // Create the error variables
    double error;
    vector<double> error_vec;

    // Current map of the temperature
    T[t] = T[t-1];

    // i = N+2; j = 1, ..., M+2 (BOTTOM)
    for(int j = 0; j < M+2; j++) {
        T[t][N+1][j] = TwBottom;
    }

    // i = 1, ..., N+1; j = M+2 (RIGHT)
    for(int i = 0; i < N+1; i++) {
        T[t][i][M+1] = TwRight_0 + TwRight_slope*mytime;
    }

    // Perform the iterations
    for(int iter = 0; iter < max_iter; iter++) {
        // Store the previous temperature
        T_unsolved = T[t];

        // i = 1; j = 2, ..., M+1 (TOP)
        for(int j = 1; j < M+1; j++) {
            T[t][0][j] = (aS[0][j]*T[t][1][j] + bP[0][j])/aP[0][j];
        }

        // i = 2, ..., N+1; j = 2, ..., M+1 (MIDDLE)
        for(int i = 1; i < N+1; i++) {
            for(int j = 1; j < M+1; j++) {
                bP[i][j] = rho[mat[i][j]]*cp[mat[i][j]]*VP/dt*T[t-1][i][j];
                T[t][i][j] = (aE[i][j]*T[t][i][j+1] + aW[i][j]*T[t][i][j-1] + aN[i][j]*T[t
    ][i-1][j] + aS[i][j]*T[t][i+1][j] + bP[i][j])/aP[i][j];
            }
```

```
        }

        // i = 2, ..., N+1; j = 1 (LEFT)
        for(int i = 1; i < N+1; i++) {
            T[t][i][0] = (aE[i][0]*T[t][i][1] + bP[i][0])/aP[i][0];
        }

        // Compute the error
        error_vec = abs_mat_to_vec(T[t] - T_unsolved);
        error = *max_element(error_vec.begin(), error_vec.end());

        // Update the temperature
        T[t] = T_unsolved + (T[t] - T_unsolved)*rf;

        // Check the convergence
        if(error < delta) {
            cout << "Converged in " << iter << " iterations! ";
            break;
        }
    }

    // i = 1; j = 1 (TOP LEFT)
    T[t][0][0] = (T[t][0][1] + T[t][1][0])/2.0;

    // Check if the solution converged
    if(error >= delta) {
        cout << "Solution did not converge!" << endl;
    }
}

// Check the energy balance
void energy_balance() {
    // Calculate the energy balance
    double Q = 0.0;
    double U = 0.0;

    // i = 1; j = 2, ..., M+1 (TOP)
    Q += qwTop*SwallL;

    // i = 2, ..., N+1; j = 1 (LEFT)
    for(int i = 1; i < N+1; i++) {
         Q += alpha_ext*Sw*(Text - T[t][i][0]);
    }

    // i = 2, ..., N+1; j = M+2 (RIGHT)
    for(int i = 1; i < N+1; i++) {
        Q += harmonic_lambda(i, M+1, "W")*(T[t][i][M+1] - T[t][i][M])/(abs(xP[M+1]-xP[M]))*
    Sw;
    }

    // i = N+2; j = 2, ..., M+2 (BOTTOM)
    for(int j = 1; j < M+1; j++) {
        Q += harmonic_lambda(N+1, j, "N")*(T[t][N+1][j] - T[t][N][j])/(abs(yP[N+1]-yP[N]))*
    Sn;
    }

    // Internal energy of every control volume
    for(int i = 1; i < N+1; i++) {
        for(int j = 1; j < M+1; j++) {
            U += rho[mat[i][j]]*cp[mat[i][j]]*VP*(T[t][i][j] - T[t-1][i][j])/dt;
        }
    }

    cout << "(U - Q)/U: " << abs((U - Q)/U) << endl;

    // Check the energy balance
```

```
    if(abs((U-Q)/U) > 1e-5) {
        wrong_time_iter += 1;
    }
}

// Solve the transitory state
void solve_transitory() {
    // Set the initial temperature
    T.resize(nt+1, vector<vector<double>>(N+2, vector<double>(M+2, T0)));

    // Energy balance
    wrong_time_iter = 0;

    // i = 1, ..., N+1; j = M+2 (RIGHT)
    for(int i = 0; i < N+1; i++) {
        T[0][i][M+1] = TwRight_0;
    }

    // i = N+2; j = 1, ..., M+2 (BOTTOM)
    for(int j = 0; j < M+2; j++) {
        T[0][N+1][j] = TwBottom;
    }

    // Perform the iterations
    for(t = 1; t <= nt; t++) {
        // Real time variable
        mytime = ti + t*dt;

        // Current time iteration
        cout << "Time step " << t << " of " << nt << ". ";

        // Solve the problem with Gauss-Seidel
        solve_with_GS();

        // Check the energy balance
        energy_balance();
    }

    // Print the energy balance
    cout << "There have been " << wrong_time_iter << " iterations where the energy balance was not satisfied!" << endl;
}

// Display results
void display_results(string method) {
    // Create output file
    ofstream file;
    file.open("output/" + method + " N=" + to_string(N) + " M=" + to_string(M) + " nt=" + to_string(int(nt)) + " results.txt");

    // Print the results
    for(int i = 0; i < N+2; i++) {
        for(int j = 0; j < M+2; j++) {
            file << T[nt][i][j] - 273 << " ";
        }
        file << endl;
    }
    file.close();
}

// Main -----------------------------------------------------------------
int main() { // Routine to run the program
    // Read the json file and assign values to the variables
    set_constants();

    // Perform initial calculations such as control volumes, discretization constants...
```

```
    previous_calculations();

    // Start time
    auto start = chrono::high_resolution_clock::now();

    // Solve the transitory state
    solve_transitory();

    // End time and duration
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;

    // Print the duration
    int minutes = int(duration.count()/60);
    double seconds = duration.count() - minutes*60;
    cout << "Elapsed time to solve transitory state: " << minutes << " minutes and " <<
   seconds << " seconds." << endl;

    // Display the results
    display_results("GS");

    return 0;
}
```

Main post-process (*Python*)

```
import json
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load data
j = json.load(open('src\\data.json'))

# Parameters
P1 = j["Geometry"]["p1"]
P2 = j["Geometry"]["p2"]
P3 = j["Geometry"]["p3"]
L = P3[0]
H = P3[1]
N = j["Control Volumes"]["N"]
M = j["Control Volumes"]["M"]
DT = j["Time Parameters"]["dt"]
TI = j["Time Parameters"]["ti"]
TF = j["Time Parameters"]["tf"]
N_T = int((TF-TI)/DT)
CMAP = 'coolwarm'
LEVELS = 15
PLOTDISTR = True

# Load data
path = 'output\\GS N=' + str(N) + ' M=' + str(M) + ' nt=' + str(N_T) + ' results.txt'
data = pd.read_csv(path, sep=r'\s+', skiprows=0, header=None)

# Convert to numpy array
data = data.values

# X axis
X = np.linspace(0, L, N+1)
X[1:] = X[1:] - 0.5*X[1]
X = np.append(X, L)

# Y axis
Y = np.linspace(H, 0, M+1)
```

```
Y[1:] = Y[1:] + 0.5*Y[M-1]
Y = np.append(Y, 0)

# Meshgrid
X, Y = np.meshgrid(X, Y)

# Surface plot
fig = plt.figure(figsize=(8, 6))
ax3D = fig.add_subplot(projection='3d')
ax3D.plot_surface(X, Y, data, cmap=CMAP)

# Set the labels
title = 'Resulting field at t = ' + str(int(TF)) + 's'
ax3D.set_title(title, fontsize=18, fontweight='bold', color='red')
ax3D.set_xlabel('X position (m)', fontsize=12, fontweight='bold', color='blue')
ax3D.set_ylabel('Y position (m)', fontsize=12, fontweight='bold', color='blue')
ax3D.set_zlabel('Temperature (Â°C)', fontsize=12, fontweight='bold', color='blue')

# Change the numbers of the axes with less values
ax3D.locator_params(axis='x', nbins=5)
ax3D.locator_params(axis='y', nbins=5)
ax3D.locator_params(axis='z', nbins=5)

# Contour plot
fig = plt.figure(figsize=(8, 6))
axC = fig.add_subplot()
contour = axC.contour(X, Y, data, cmap=CMAP, levels=LEVELS)
plt.clabel(contour, colors = 'k', fmt = '%2.1fÂ°C', fontsize=10, inline=True)
contour_filled = axC.contourf(X, Y, data, cmap=CMAP, levels=LEVELS, alpha=0.8)

# Aspect and drawings
axC.set_aspect(1)
plt.plot([P1[0], P1[0]], [0, H], color='black', linestyle='dashed', linewidth=0.5)
plt.plot([P1[0], L], [P2[1], P2[1]], color='black', linestyle='dashed', linewidth=0.5)
plt.plot([0, P1[0]], [P1[1], P1[1]], color='black', linestyle='dashed', linewidth=0.5)

# Set the labels
title = 'Resulting field at t = ' + str(int(TF)) + 's'
axC.set_title(title, fontsize=18, fontweight='bold')
axC.set_xlabel('X position (m)', fontsize=12)
axC.set_ylabel('Y position (m)', fontsize=12)

# Change the numbers of the axes with less values
axC.locator_params(axis='x', nbins=5)
axC.locator_params(axis='y', nbins=5)

# Field plot
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot()
im = ax.imshow(data, cmap=CMAP)

# Set the labels
ax.set_title(title, fontsize=18, fontweight='bold')
ax.set_xlabel('X position (m)', fontsize=12)
ax.set_ylabel('Y position (m)', fontsize=12)

# Aspect and drawings
ax.set_aspect(H/L)
plt.plot([(N+1)*P1[0]/L, (N + 1)*P1[0]/L], [0, M+1], color='black', linestyle='dashed')
plt.plot([(N+1)*P1[0]/L, N+1], [M+1-(M+2)*P2[1]/H, M+1-(M+2)*P2[1]/H], color='black',
    linestyle='dashed')
plt.plot([0, (N+1)*P1[0]/L], [M+1-(M+2)*P1[1]/H, M+1-(M+2)*P1[1]/H], color='black',
    linestyle='dashed')

# Tick labels from 0 to L
ax.set_xticks(np.linspace(0, N+1, 2))
```

```
ax.set_xticklabels(np.linspace(0, L, 2))

# Tick labels from 0 to H
ax.set_yticks(np.linspace(0, M+1, 2))
ax.set_yticklabels(np.linspace(H, 0, 2))

# Add a colorbar
fig.colorbar(im, ax=ax, label='Temperature (Â°C)', ticks=np.linspace(data.min(), data.max()
    , 5))

# Plot the mesh
if(PLOTDISTR):
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot()
    ones = np.ones((M+2, N+2))
    im = ax.imshow(ones, cmap=CMAP)

    # Set the labels
    ax.set_title('Material distribution', fontsize=18, fontweight='bold')
    ax.set_xlabel('X position (m)', fontsize=12)
    ax.set_ylabel('Y position (m)', fontsize=12)

    # Aspect and drawings
    ax.set_aspect(H/L)
    plt.plot([(N+1)*P1[0]/L, (N + 1)*P1[0]/L], [0, M+1], color='black', linestyle='dashed')
    plt.plot([(N+1)*P1[0]/L, N+1], [M+1-(M+2)*P2[1]/H, M+1-(M+2)*P2[1]/H], color='black',
    linestyle='dashed')
    plt.plot([0, (N+1)*P1[0]/L], [M+1-(M+2)*P1[1]/H, M+1-(M+2)*P1[1]/H], color='black',
    linestyle='dashed')

    # Tick labels from 0 to L
    ax.set_xticks(np.linspace(0, N+1, 2))
    ax.set_xticklabels(np.linspace(0, L, 2))

    # Tick labels from 0 to H
    ax.set_yticks(np.linspace(0, M+1, 2))
    ax.set_yticklabels(np.linspace(H, 0, 2))

plt.show()
```

Numerical study post-process (*Python*)

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

# Parameters
FINAL_TIME = 500

# Elements study data
time_el = np.array([4.05, 11.36, 28.59, 53.74, 112.62, 174.76])
iter_el = np.array([19, 31, 45, 61, 79, 100])
elements = np.array([40, 60, 80, 100, 120, 140])

space_el = np.linspace(elements[0], elements[-1], 100)
reg = np.polyfit(elements, time_el, 3)
poly_el = np.poly1d(reg)

print(poly_el)

# Relaxation factor study data
time_rf = np.array([79.51, 44.13, 29.30, 23.27, 22.84])
iter_rf = np.array([122, 67, 45, 36, 32])
relax_factor = np.array([0.4, 0.7, 1, 1.2, 1.4])

```

```python
space_rf = np.linspace(relax_factor[0], relax_factor[-1], 100)
reg = np.polyfit(relax_factor, time_rf, 3)
poly_rf = np.poly1d(reg)

print(poly_rf)

# Precision study data
time_pr = np.array([16.29, 19.70, 24.29, 28.55, 31.35, 40.16])
iter_pr = np.array([21, 29, 37, 45, 53, 61])
precision = np.array(np.log10([1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]))

space_pr = np.linspace(precision[0], precision[-1], 100)
reg = np.polyfit(precision, time_pr, 1)
poly_pr = np.poly1d(reg)

print(poly_pr)

# Plot: Elements vs Time and Iterations
figure, ax1 = plt.subplots()
ax1.plot(elements, time_el, 'o-', color='blue')
ax1.plot(space_el, poly_el(space_el), 'b--')
ax1.legend(['Data', 'Polynomial regression'], loc='upper left')
ax1.set_xlabel('Number of elements', fontsize=12)
ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
ax1.tick_params(axis='y', labelcolor='blue')
for i in range(len(elements)):
    ax1.text(elements[i] + 3, time_el[i] - 2, str(time_el[i]), fontsize=9, color='blue', ha
    ='left', va='top')

ax2 = ax1.twinx()
ax2.plot(elements, iter_el, 'o-', color='red')
ax2.set_ylabel('Iterations', fontsize=12, color='red')
ax2.tick_params(axis='y', labelcolor='red')
for i in range(len(elements)):
    ax2.text(elements[i] - 1.2, iter_el[i] + 0.1, str(iter_el[i]), fontsize=9, color='red',
     ha='right', va='bottom')

title = 'Resolution until t=' + str(FINAL_TIME) + 's'
plt.title(title, fontsize=18, fontweight='bold')

# Plot: Relaxation factor vs Time and Iterations
figure, ax1 = plt.subplots()
ax1.plot(relax_factor, time_rf, 'o-', color='blue')
ax1.plot(space_rf, poly_rf(space_rf), 'b--')
ax1.legend(['Data', 'Polynomial regression'], loc='upper right')
ax1.set_xlabel('Relaxation factor', fontsize=12)
ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
ax1.tick_params(axis='y', labelcolor='blue')
for i in range(len(relax_factor)):
    ax1.text(relax_factor[i] - 0.08, time_rf[i] - 1.5, str(time_rf[i]), fontsize=9, color='
    blue', ha='left', va='top')

ax2 = ax1.twinx()
ax2.plot(relax_factor, iter_rf, 'o-', color='red')
ax2.set_ylabel('Iterations', fontsize=12, color='red')
ax2.tick_params(axis='y', labelcolor='red')
for i in range(len(relax_factor)):
    ax2.text(relax_factor[i] + 0.04, iter_rf[i] + 1.5, str(iter_rf[i]), fontsize=9, color='
    red', ha='right', va='bottom')

title = 'Resolution until t=' + str(FINAL_TIME) + 's'
plt.title(title, fontsize=18, fontweight='bold')

# Plot: Precision vs Time and Iterations
figure, ax1 = plt.subplots()
ax1.plot(precision, time_pr, 'o-', color='blue')
```

```
ax1.plot(space_pr, poly_pr(space_pr), 'b--')
ax1.legend(['Data', 'Polynomial regression'], loc='upper right')
ax1.set_xlabel('log($\\delta$) (K)', fontsize=12)
ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
ax1.tick_params(axis='y', labelcolor='blue')
for i in range(len(precision)):
    ax1.text(precision[i] - 0.5, time_pr[i] - 0.5, str(time_pr[i]), fontsize=9, color='blue', ha='left', va='top')

ax2 = ax1.twinx()
ax2.plot(precision, iter_pr, 'o-', color='red')
ax2.set_ylabel('Iterations', fontsize=12, color='red')
ax2.tick_params(axis='y', labelcolor='red')
for i in range(len(precision)):
    ax2.text(precision[i] + 0.2, iter_pr[i] + 0.5, str(iter_pr[i]), fontsize=9, color='red', ha='right', va='bottom')

title = 'Resolution until t=' + str(FINAL_TIME) + 's'
plt.title(title, fontsize=18, fontweight='bold')

plt.show()
```