

# 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++)

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <vector>
5 #include <chrono>
6 #include <algorithm>
7 #include "json.hpp"
8
9 using namespace std;
10 using json = nlohmann::json;
11
12 // Variables -----
13 // Define all the variables, including the j object to read the json file
14 json j;
15 int i, t, N, M, max_iter, wrong_time_iter;
16 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;
17 vector<double> p1, p2, p3, rho, cp, lambda, xP, yP;
18 vector<vector<int>>> mat;
19 vector<vector<double>>> aP, aE, aW, aN, aS, bP;
20 vector<vector<vector<double>>>> T;
21
22 // Functions -----
23 // Overload the - operator for matrices
24 vector<vector<double>>> operator-(const vector<vector<double>>>& a, const vector<vector<
    double>>>& b) {
25     // Create the result matrix
26     vector<vector<double>>> result = a;
27
28     // Subtract the matrices
29     for(size_t i = 0; i < a.size(); i++) {
30         for(size_t j = 0; j < a[i].size(); j++) {
31             result[i][j] -= b[i][j];
32         }
33     }
34
35     return result;
36 }
37
38 // Overload the + operator for matrices
39 vector<vector<double>>> operator+(const vector<vector<double>>>& a, const vector<vector<
    double>>>& b) {
40     // Create the result matrix
41     vector<vector<double>>> result = a;
42
43     // Add the matrices
44     for(size_t i = 0; i < a.size(); i++) {
45         for(size_t j = 0; j < a[i].size(); j++) {
46             result[i][j] += b[i][j];
```

```

47     }
48 }
49
50 return result;
51 }
52
53 // Overload the * operator for matrices times scalars
54 vector<vector<double>> operator*(const vector<vector<double>>& a, const double& b) {
55     // Create the result matrix
56     vector<vector<double>> result = a;
57
58     // Multiply the matrix by the scalar
59     for(size_t i = 0; i < a.size(); i++) {
60         for(size_t j = 0; j < a[i].size(); j++) {
61             result[i][j] *= b;
62         }
63     }
64
65     return result;
66 }
67
68 // Absolute value of matrix elements stored in a vector
69 vector<double> abs_mat_to_vec(const vector<vector<double>>& a) {
70     // Create the result matrix
71     vector<double> result;
72
73     // Calculate the absolute value of the matrix
74     for(size_t i = 0; i < a.size(); i++) {
75         for(size_t j = 0; j < a[i].size(); j++) {
76             result.push_back(abs(a[i][j]));
77         }
78     }
79
80     return result;
81 }
82
83 // Get the correct lambda
84 double harmonic_lambda(int a, int b, string position) {
85     // Check the position and return the correct lambda
86     if(position == "E") {
87         return 2*lambda[mat[a][b]]*lambda[mat[a][b+1]]/(lambda[mat[a][b]] + lambda[mat[a][b
88 +1]]);
89     } else if(position == "W") {
90         return 2*lambda[mat[a][b-1]]*lambda[mat[a][b]]/(lambda[mat[a][b-1]] + lambda[mat[a
91 ][b]]);
92     } else if(position == "N") {
93         return 2*lambda[mat[a][b]]*lambda[mat[a-1][b]]/(lambda[mat[a][b]] + lambda[mat[a
94 -1][b]]);
95     } else if(position == "S") {
96         return 2*lambda[mat[a+1][b]]*lambda[mat[a][b]]/(lambda[mat[a+1][b]] + lambda[mat[a
97 ][b]]);
98     } else {
99         cout << "Invalid position!" << endl;
100         exit(0);
101         return 0.0;
102     }
103 }
104
105 // Actions -----
106 // Assign the constants from the json file to the variables
107 void set_constants() {
108     // Read the file
109     ifstream json_file("src/data.json");
110     json_file >> j;
111
112     // Set the Geometry

```

```

109 p1.resize(2, 0.0);
110 p2.resize(2, 0.0);
111 p3.resize(2, 0.0);
112 p1 = j["Geometry"]["p1"].get<vector<double>>();
113 p2 = j["Geometry"]["p2"].get<vector<double>>();
114 p3 = j["Geometry"]["p3"].get<vector<double>>();
115
116 // General dimensions
117 L = p3[0];
118 H = p3[1];
119
120 // Set the Material Properties
121 rho.resize(4, 0.0);
122 cp.resize(4, 0.0);
123 lambda.resize(4, 0.0);
124 for(i = 0; i < 4; i++) {
125     rho[i] = j["Material Properties"][i]["rho"];
126     cp[i] = j["Material Properties"][i]["cp"];
127     lambda[i] = j["Material Properties"][i]["lambda"];
128 }
129
130 // Set the Boundary Conditions
131 TwBottom = j["Boundary Conditions"]["TwBottom"];
132 TwRight_0 = j["Boundary Conditions"]["TwRight"][0];
133 TwRight_slope = j["Boundary Conditions"]["TwRight"][1];
134 qwTop = j["Boundary Conditions"]["qwTop"];
135 Text = j["Boundary Conditions"]["Text"];
136 alpha_ext = j["Boundary Conditions"]["alpha_ext"];
137
138 // Set the Control Volumes
139 N = j["Control Volumes"]["N"];
140 M = j["Control Volumes"]["M"];
141 dx = L/M;
142 dy = H/N;
143
144 // Set the Time Parameters
145 dt = j["Time Parameters"]["dt"];
146 ti = j["Time Parameters"]["ti"];
147 tf = j["Time Parameters"]["tf"];
148 nt = (tf-ti)/dt;
149
150 // Set the Solver Parameters
151 T0 = j["Solver Parameters"]["T0"];
152 rf = j["Solver Parameters"]["Relaxation Factor"];
153 max_iter = j["Solver Parameters"]["Maximum Iterations"];
154 delta = j["Solver Parameters"]["delta"];
155 }
156
157 // Find the material properties for each node
158 void find_materials() {
159     // Resize the matrix
160     mat.resize(N+2, vector<int>(M+2, 0));
161
162     // Find the materials
163     for(int i = 0; i < N+2; i++) {
164         for(int j = 0; j < M+2; j++) {
165             if(xP[j] <= p1[0] && yP[i] <= p1[1]) {
166                 mat[i][j] = 0;
167             } else if(xP[j] > p1[0] && yP[i] <= p2[1]) {
168                 mat[i][j] = 1;
169             } else if(xP[j] <= p2[0]) {
170                 mat[i][j] = 2;
171             } else {
172                 mat[i][j] = 3;
173             }
174         }
175     }
176 }

```

```

175 }
176
177 // Create the materials file
178 ofstream mater;
179 mater.open("output/materials.txt");
180
181 // Print the materials
182 for(int i = 0; i < N+2; i++) {
183     for(int j = 0; j < M+2; j++) {
184         mater << mat[i][j] << " ";
185     }
186     mater << endl;
187 }
188 mater.close();
189 }
190
191 // Set the discretization constants that do not change over time
192 void discretization_constants() {
193     // Initialize values
194     aP.resize(N+2, vector<double>(M+2, 0.0));
195     aE.resize(N+2, vector<double>(M+2, 0.0));
196     aW.resize(N+2, vector<double>(M+2, 0.0));
197     aN.resize(N+2, vector<double>(M+2, 0.0));
198     aS.resize(N+2, vector<double>(M+2, 0.0));
199     bP.resize(N+2, vector<double>(M+2, 0.0));
200
201     // i = 1; j = 2, ..., M+1 (TOP)
202     for(int j = 1; j < M+1; j++) {
203         aS[0][j] = harmonic_lambda(0, j, "S")/(abs(yP[0]-yP[1]))*Ss;
204         aP[0][j] = aS[0][j];
205         bP[0][j] = qwTop*Sn;
206     }
207
208     // i = 2, ..., N+1; j = 2, ..., M+1 (MIDDLE)
209     for(int i = 1; i < N+1; i++) {
210         for(int j = 1; j < M+1; j++) {
211             aW[i][j] = harmonic_lambda(i, j, "W")/(abs(xP[j]-xP[j-1]))*Sw;
212             aE[i][j] = harmonic_lambda(i, j, "E")/(abs(xP[j+1]-xP[j]))*Se;
213             aN[i][j] = harmonic_lambda(i, j, "N")/(abs(yP[i]-yP[i-1]))*Sn;
214             aS[i][j] = harmonic_lambda(i, j, "S")/(abs(yP[i+1]-yP[i]))*Ss;
215             aP[i][j] = aE[i][j] + aW[i][j] + aN[i][j] + aS[i][j] + rho[mat[i][j]]*cp[mat[i]
216 ]][j]]*VP/dt;
217         }
218     }
219
220     // i = 2, ..., N+1; j = 1 (LEFT)
221     for(int i = 1; i < N+1; i++) {
222         aE[i][0] = harmonic_lambda(i, 0, "E")/(abs(xP[1]-xP[0]))*Se;
223         aP[i][0] = aE[i][0] + alpha_ext*Sw;
224         bP[i][0] = alpha_ext*Sw*Text;
225     }
226 }
227
228 // Perform initial calculations
229 void previous_calculations() {
230     // Geometry properties
231     V = L*H;
232     SwallH = H;
233     SwallL = L;
234     xP.resize(M+2, 0.0);
235     yP.resize(N+2, 0.0);
236     xP[0] = 0.0;
237     yP[0] = H;
238     xP[1] = dx/2;
239     yP[1] = H - dy/2;
240     xP[M+1] = L;

```

```

240 yP[N+1] = 0.0;
241
242 // Control volume x positions
243 for(i = 2; i < M+1; i++) {
244     xP[i] = xP[i-1] + dx;
245 }
246
247 // Control volume y positions
248 for(i = 2; i < N+1; i++) {
249     yP[i] = yP[i-1] - dy;
250 }
251
252 // Control volume properties (structured mesh)
253 VP = dx*dy;
254 Sn = dx;
255 Ss = dx;
256 Sw = dy;
257 Se = dy;
258
259 // Find material properties for each node
260 find_materials();
261
262 // Set the discretization constants that do not change over time
263 discretization_constants();
264 }
265
266 // Solve the problem with Gauss-Seidel
267 void solve_with_GS() {
268     // Create the T_unsolved matrix
269     vector<vector<double>> T_unsolved;
270     T_unsolved.resize(N+2, vector<double>(M+2, T0));
271
272     // Create the error variables
273     double error;
274     vector<double> error_vec;
275
276     // Current map of the temperature
277     T[t] = T[t-1];
278
279     // i = N+2; j = 1, ..., M+2 (BOTTOM)
280     for(int j = 0; j < M+2; j++) {
281         T[t][N+1][j] = TwBottom;
282     }
283
284     // i = 1, ..., N+1; j = M+2 (RIGHT)
285     for(int i = 0; i < N+1; i++) {
286         T[t][i][M+1] = TwRight_0 + TwRight_slope*mytime;
287     }
288
289     // Perform the iterations
290     for(int iter = 0; iter < max_iter; iter++) {
291         // Store the previous temperature
292         T_unsolved = T[t];
293
294         // i = 1; j = 2, ..., M+1 (TOP)
295         for(int j = 1; j < M+1; j++) {
296             T[t][0][j] = (aS[0][j]*T[t][1][j] + bP[0][j])/aP[0][j];
297         }
298
299         // i = 2, ..., N+1; j = 2, ..., M+1 (MIDDLE)
300         for(int i = 1; i < N+1; i++) {
301             for(int j = 1; j < M+1; j++) {
302                 bP[i][j] = rho[mat[i][j]]*cp[mat[i][j]]*VP/dt*T[t-1][i][j];
303                 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];
304             }

```

```

305     }
306
307     // i = 2, ..., N+1; j = 1 (LEFT)
308     for(int i = 1; i < N+1; i++) {
309         T[t][i][0] = (aE[i][0]*T[t][i][1] + bP[i][0])/aP[i][0];
310     }
311
312     // Compute the error
313     error_vec = abs_mat_to_vec(T[t] - T_unsolved);
314     error = *max_element(error_vec.begin(), error_vec.end());
315
316     // Update the temperature
317     T[t] = T_unsolved + (T[t] - T_unsolved)*rf;
318
319     // Check the convergence
320     if(error < delta) {
321         cout << "Converged in " << iter << " iterations! ";
322         break;
323     }
324 }
325
326 // i = 1; j = 1 (TOP LEFT)
327 T[t][0][0] = (T[t][0][1] + T[t][1][0])/2.0;
328
329 // Check if the solution converged
330 if(error >= delta) {
331     cout << "Solution did not converge!" << endl;
332 }
333 }
334
335 // Check the energy balance
336 void energy_balance() {
337     // Calculate the energy balance
338     double Q = 0.0;
339     double U = 0.0;
340
341     // i = 1; j = 2, ..., M+1 (TOP)
342     Q += qwTop*SwallL;
343
344     // i = 2, ..., N+1; j = 1 (LEFT)
345     for(int i = 1; i < N+1; i++) {
346         Q += alpha_ext*Sw*(Text - T[t][i][0]);
347     }
348
349     // i = 2, ..., N+1; j = M+2 (RIGHT)
350     for(int i = 1; i < N+1; i++) {
351         Q += harmonic_lambda(i, M+1, "W")*(T[t][i][M+1] - T[t][i][M])/(abs(xP[M+1]-xP[M]))*
Sw;
352     }
353
354     // i = N+2; j = 2, ..., M+2 (BOTTOM)
355     for(int j = 1; j < M+1; j++) {
356         Q += harmonic_lambda(N+1, j, "N")*(T[t][N+1][j] - T[t][N][j])/(abs(yP[N+1]-yP[N]))*
Sn;
357     }
358
359     // Internal energy of every control volume
360     for(int i = 1; i < N+1; i++) {
361         for(int j = 1; j < M+1; j++) {
362             U += rho[mat[i][j]]*cp[mat[i][j]]*VP*(T[t][i][j] - T[t-1][i][j])/dt;
363         }
364     }
365
366     cout << "(U - Q)/U: " << abs((U - Q)/U) << endl;
367
368     // Check the energy balance

```

```

369     if(abs((U-Q)/U) > 1e-5) {
370         wrong_time_iter += 1;
371     }
372 }
373
374 // Solve the transitory state
375 void solve_transitory() {
376     // Set the initial temperature
377     T.resize(nt+1, vector<vector<double>>(N+2, vector<double>(M+2, T0)));
378
379     // Energy balance
380     wrong_time_iter = 0;
381
382     // i = 1, ..., N+1; j = M+2 (RIGHT)
383     for(int i = 0; i < N+1; i++) {
384         T[0][i][M+1] = TwRight_0;
385     }
386
387     // i = N+2; j = 1, ..., M+2 (BOTTOM)
388     for(int j = 0; j < M+2; j++) {
389         T[0][N+1][j] = TwBottom;
390     }
391
392     // Perform the iterations
393     for(t = 1; t <= nt; t++) {
394         // Real time variable
395         mytime = ti + t*dt;
396
397         // Current time iteration
398         cout << "Time step " << t << " of " << nt << ". ";
399
400         // Solve the problem with Gauss-Seidel
401         solve_with_GS();
402
403         // Check the energy balance
404         energy_balance();
405     }
406
407     // Print the energy balance
408     cout << "There have been " << wrong_time_iter << " iterations where the energy balance
409     was not satisfied!" << endl;
410 }
411
412 // Display results
413 void display_results(string method) {
414     // Create output file
415     ofstream file;
416     file.open("output/" + method + " N=" + to_string(N) + " M=" + to_string(M) + " nt=" +
417     to_string(int(nt)) + " results.txt");
418
419     // Print the results
420     for(int i = 0; i < N+2; i++) {
421         for(int j = 0; j < M+2; j++) {
422             file << T[nt][i][j] - 273 << " ";
423         }
424         file << endl;
425     }
426     file.close();
427 }
428
429 // Main -----
430 int main() { // Routine to run the program
431     // Read the json file and assign values to the variables
432     set_constants();
433
434     // Perform initial calculations such as control volumes, discretization constants...

```

```

433 previous_calculations();
434
435 // Start time
436 auto start = chrono::high_resolution_clock::now();
437
438 // Solve the transitory state
439 solve_transitory();
440
441 // End time and duration
442 auto end = chrono::high_resolution_clock::now();
443 chrono::duration<double> duration = end - start;
444
445 // Print the duration
446 int minutes = int(duration.count()/60);
447 double seconds = duration.count() - minutes*60;
448 cout << "Elapsed time to solve transitory state: " << minutes << " minutes and " <<
seconds << " seconds." << endl;
449
450 // Display the results
451 display_results("GS");
452
453 return 0;
454 }

```

## Main post-process (Python)

```

1 import json
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5
6 # Load data
7 j = json.load(open('src\\data.json'))
8
9 # Parameters
10 P1 = j["Geometry"]["p1"]
11 P2 = j["Geometry"]["p2"]
12 P3 = j["Geometry"]["p3"]
13 L = P3[0]
14 H = P3[1]
15 N = j["Control Volumes"]["N"]
16 M = j["Control Volumes"]["M"]
17 DT = j["Time Parameters"]["dt"]
18 TI = j["Time Parameters"]["ti"]
19 TF = j["Time Parameters"]["tf"]
20 N_T = int((TF-TI)/DT)
21 CMAP = 'coolwarm'
22 LEVELS = 15
23 PLOTDISTR = True
24
25 # Load data
26 path = 'output\\GS N=' + str(N) + ' M=' + str(M) + ' nt=' + str(N_T) + ' results.txt'
27 data = pd.read_csv(path, sep=r'\s+', skiprows=0, header=None)
28
29 # Convert to numpy array
30 data = data.values
31
32 # X axis
33 X = np.linspace(0, L, N+1)
34 X[1:] = X[1:] - 0.5*X[1]
35 X = np.append(X, L)
36
37 # Y axis
38 Y = np.linspace(H, 0, M+1)

```



```

39 Y[1:] = Y[1:] + 0.5*Y[M-1]
40 Y = np.append(Y, 0)
41
42 # Meshgrid
43 X, Y = np.meshgrid(X, Y)
44
45 # Surface plot
46 fig = plt.figure(figsize=(8, 6))
47 ax3D = fig.add_subplot(projection='3d')
48 ax3D.plot_surface(X, Y, data, cmap=CMAP)
49
50 # Set the labels
51 title = 'Resulting field at t = ' + str(int(TF)) + 's'
52 ax3D.set_title(title, fontsize=18, fontweight='bold', color='red')
53 ax3D.set_xlabel('X position (m)', fontsize=12, fontweight='bold', color='blue')
54 ax3D.set_ylabel('Y position (m)', fontsize=12, fontweight='bold', color='blue')
55 ax3D.set_zlabel('Temperature (°C)', fontsize=12, fontweight='bold', color='blue')
56
57 # Change the numbers of the axes with less values
58 ax3D.locator_params(axis='x', nbins=5)
59 ax3D.locator_params(axis='y', nbins=5)
60 ax3D.locator_params(axis='z', nbins=5)
61
62 # Contour plot
63 fig = plt.figure(figsize=(8, 6))
64 axC = fig.add_subplot()
65 contour = axC.contour(X, Y, data, cmap=CMAP, levels=LEVELS)
66 plt.clabel(contour, colors = 'k', fmt = '%2.1f°C', fontsize=10, inline=True)
67 contour_filled = axC.contourf(X, Y, data, cmap=CMAP, levels=LEVELS, alpha=0.8)
68
69 # Aspect and drawings
70 axC.set_aspect(1)
71 plt.plot([P1[0], P1[0]], [0, H], color='black', linestyle='dashed', linewidth=0.5)
72 plt.plot([P1[0], L], [P2[1], P2[1]], color='black', linestyle='dashed', linewidth=0.5)
73 plt.plot([0, P1[0]], [P1[1], P1[1]], color='black', linestyle='dashed', linewidth=0.5)
74
75 # Set the labels
76 title = 'Resulting field at t = ' + str(int(TF)) + 's'
77 axC.set_title(title, fontsize=18, fontweight='bold')
78 axC.set_xlabel('X position (m)', fontsize=12)
79 axC.set_ylabel('Y position (m)', fontsize=12)
80
81 # Change the numbers of the axes with less values
82 axC.locator_params(axis='x', nbins=5)
83 axC.locator_params(axis='y', nbins=5)
84
85 # Field plot
86 fig = plt.figure(figsize=(8, 6))
87 ax = fig.add_subplot()
88 im = ax.imshow(data, cmap=CMAP)
89
90 # Set the labels
91 ax.set_title(title, fontsize=18, fontweight='bold')
92 ax.set_xlabel('X position (m)', fontsize=12)
93 ax.set_ylabel('Y position (m)', fontsize=12)
94
95 # Aspect and drawings
96 ax.set_aspect(H/L)
97 plt.plot([(N+1)*P1[0]/L, (N + 1)*P1[0]/L], [0, M+1], color='black', linestyle='dashed')
98 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',
99          linestyle='dashed')
100 plt.plot([0, (N+1)*P1[0]/L], [M+1-(M+2)*P1[1]/H, M+1-(M+2)*P1[1]/H], color='black',
101          linestyle='dashed')
102
103 # Tick labels from 0 to L
104 ax.set_xticks(np.linspace(0, N+1, 2))

```

```

103 ax.set_xticklabels(np.linspace(0, L, 2))
104
105 # Tick labels from 0 to H
106 ax.set_yticks(np.linspace(0, M+1, 2))
107 ax.set_yticklabels(np.linspace(H, 0, 2))
108
109 # Add a colorbar
110 fig.colorbar(im, ax=ax, label='Temperature (°C)', ticks=np.linspace(data.min(), data.max(),
    , 5))
111
112 # Plot the mesh
113 if(PLOTDISTR):
114     fig = plt.figure(figsize=(8, 6))
115     ax = fig.add_subplot()
116     ones = np.ones((M+2, N+2))
117     im = ax.imshow(ones, cmap=CMAP)
118
119     # Set the labels
120     ax.set_title('Material distribution', fontsize=18, fontweight='bold')
121     ax.set_xlabel('X position (m)', fontsize=12)
122     ax.set_ylabel('Y position (m)', fontsize=12)
123
124     # Aspect and drawings
125     ax.set_aspect(H/L)
126     plt.plot([(N+1)*P1[0]/L, (N + 1)*P1[0]/L], [0, M+1], color='black', linestyle='dashed')
127     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')
128     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')
129
130     # Tick labels from 0 to L
131     ax.set_xticks(np.linspace(0, N+1, 2))
132     ax.set_xticklabels(np.linspace(0, L, 2))
133
134     # Tick labels from 0 to H
135     ax.set_yticks(np.linspace(0, M+1, 2))
136     ax.set_yticklabels(np.linspace(H, 0, 2))
137
138 plt.show()

```

## Numerical study post-process (Python)

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Parameters
5 FINAL_TIME = 500
6
7 # Elements study data
8 time_el = np.array([4.05, 11.36, 28.59, 53.74, 112.62, 174.76])
9 iter_el = np.array([19, 31, 45, 61, 79, 100])
10 elements = np.array([40, 60, 80, 100, 120, 140])
11
12 space_el = np.linspace(elements[0], elements[-1], 100)
13 reg = np.polyfit(elements, time_el, 3)
14 poly_el = np.poly1d(reg)
15
16 print(poly_el)
17
18 # Relaxation factor study data
19 time_rf = np.array([79.51, 44.13, 29.30, 23.27, 22.84])
20 iter_rf = np.array([122, 67, 45, 36, 32])
21 relax_factor = np.array([0.4, 0.7, 1, 1.2, 1.4])
22

```

```

23 space_rf = np.linspace(relax_factor[0], relax_factor[-1], 100)
24 reg = np.polyfit(relax_factor, time_rf, 3)
25 poly_rf = np.poly1d(reg)
26
27 print(poly_rf)
28
29 # Precision study data
30 time_pr = np.array([16.29, 19.70, 24.29, 28.55, 31.35, 40.16])
31 iter_pr = np.array([21, 29, 37, 45, 53, 61])
32 precision = np.array(np.log10([1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]))
33
34 space_pr = np.linspace(precision[0], precision[-1], 100)
35 reg = np.polyfit(precision, time_pr, 1)
36 poly_pr = np.poly1d(reg)
37
38 print(poly_pr)
39
40 # Plot: Elements vs Time and Iterations
41 figure, ax1 = plt.subplots()
42 ax1.plot(elements, time_el, 'o-', color='blue')
43 ax1.plot(space_el, poly_el(space_el), 'b--')
44 ax1.legend(['Data', 'Polynomial regression'], loc='upper left')
45 ax1.set_xlabel('Number of elements', fontsize=12)
46 ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
47 ax1.tick_params(axis='y', labelcolor='blue')
48 for i in range(len(elements)):
49     ax1.text(elements[i] + 3, time_el[i] - 2, str(time_el[i]), fontsize=9, color='blue', ha
50             = 'left', va='top')
51
52 ax2 = ax1.twinx()
53 ax2.plot(elements, iter_el, 'o-', color='red')
54 ax2.set_ylabel('Iterations', fontsize=12, color='red')
55 ax2.tick_params(axis='y', labelcolor='red')
56 for i in range(len(elements)):
57     ax2.text(elements[i] - 1.2, iter_el[i] + 0.1, str(iter_el[i]), fontsize=9, color='red',
58             ha='right', va='bottom')
59
60 title = 'Resolution until t=' + str(FINAL_TIME) + 's'
61 plt.title(title, fontsize=18, fontweight='bold')
62
63 # Plot: Relaxation factor vs Time and Iterations
64 figure, ax1 = plt.subplots()
65 ax1.plot(relax_factor, time_rf, 'o-', color='blue')
66 ax1.plot(space_rf, poly_rf(space_rf), 'b--')
67 ax1.legend(['Data', 'Polynomial regression'], loc='upper right')
68 ax1.set_xlabel('Relaxation factor', fontsize=12)
69 ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
70 ax1.tick_params(axis='y', labelcolor='blue')
71 for i in range(len(relax_factor)):
72     ax1.text(relax_factor[i] - 0.08, time_rf[i] - 1.5, str(time_rf[i]), fontsize=9, color='
73             blue', ha='left', va='top')
74
75 ax2 = ax1.twinx()
76 ax2.plot(relax_factor, iter_rf, 'o-', color='red')
77 ax2.set_ylabel('Iterations', fontsize=12, color='red')
78 ax2.tick_params(axis='y', labelcolor='red')
79 for i in range(len(relax_factor)):
80     ax2.text(relax_factor[i] + 0.04, iter_rf[i] + 1.5, str(iter_rf[i]), fontsize=9, color='
81             red', ha='right', va='bottom')
82
83 title = 'Resolution until t=' + str(FINAL_TIME) + 's'
84 plt.title(title, fontsize=18, fontweight='bold')
85
86 # Plot: Precision vs Time and Iterations
87 figure, ax1 = plt.subplots()
88 ax1.plot(precision, time_pr, 'o-', color='blue')

```

---

```

85 ax1.plot(space_pr, poly_pr(space_pr), 'b--')
86 ax1.legend(['Data', 'Polynomial regression'], loc='upper right')
87 ax1.set_xlabel('log( $\Delta$ ) (K)', fontsize=12)
88 ax1.set_ylabel('Time (s)', fontsize=12, color='blue')
89 ax1.tick_params(axis='y', labelcolor='blue')
90 for i in range(len(precision)):
91     ax1.text(precision[i] - 0.5, time_pr[i] - 0.5, str(time_pr[i]), fontsize=9, color='blue',
92             ha='left', va='top')
93
94 ax2 = ax1.twinx()
95 ax2.plot(precision, iter_pr, 'o-', color='red')
96 ax2.set_ylabel('Iterations', fontsize=12, color='red')
97 ax2.tick_params(axis='y', labelcolor='red')
98 for i in range(len(precision)):
99     ax2.text(precision[i] + 0.2, iter_pr[i] + 0.5, str(iter_pr[i]), fontsize=9, color='red',
100            ha='right', va='bottom')
101
102 title = 'Resolution until t=' + str(FINAL_TIME) + 's'
103 plt.title(title, fontsize=18, fontweight='bold')
104
105 plt.show()

```