POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Nonlinear Springs and Boundary Element Method

Final Report of
Advanced Computational Mechanics - G. Novati

Master Degree in
Mathematical Engineering

Author: **Matteo Bonfadini**

Date: 25/07/2024
Academic Year: 2023-24

Contents

1 | Systems of Nonlinear Springs

1.1. Introduction

Many engineering applications show nonlinear behaviours, and linear systems cannot provide an acceptable solution in more and more situations. In solid mechanics, such a situation usually occurs when the deformation is large, material response is complex, boundary conditions vary, etc. In this context, linear systems could be seen as approximation of nonlinear systems under limited conditions, e.g., with small deformation.

In general there is no analytical way of finding the solution of a system of nonlinear equations, and this explains why computational sciences have become increasingly in demand.

In the following section we discuss a variation of a problem presented during the course and in the book [2].

1.2. System of Nonlinear Springs

Consider three non-linear springs connected in series and in parallel, as shown in Figure 1.1.

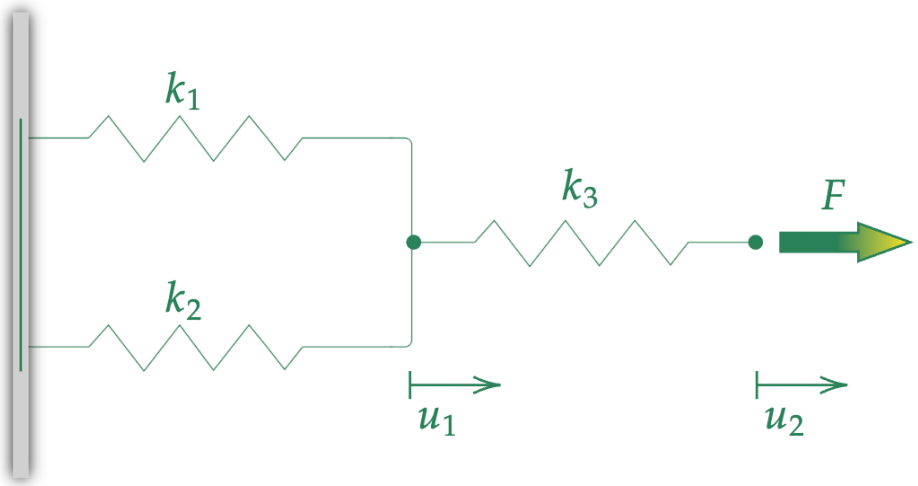


Figure 1.1: System of nonlinear springs.

The stiffness of the springs depends on the elongation of springs such that

$$k_i = k_i\left(e_i\right) = \alpha_i + \beta_i \cdot e_i \quad [\mathrm{N/mm}], \qquad\qquad i = 1, 2, 3,$$

with e_i being the elongation of the i-th spring and coefficients (α_i, β_i) being listed in Table 1.2. In this way, although the geometry of the problem is linear, it is ruled by nonlinear costitutive laws.

In Figure 1.1 is also shown the load application $F = 100$ [N] at the tip.

Our interest lies in calculating u_1 and u_2, the nodal displacements at the two nodes.

i	α_i	β_i
1	500	50
2	200	100
3	500	100

Table 1.1: Springs coefficients.

From Figure 1.2 one can notice that the stiffness (the local tangent) is increasing for increased elongations.

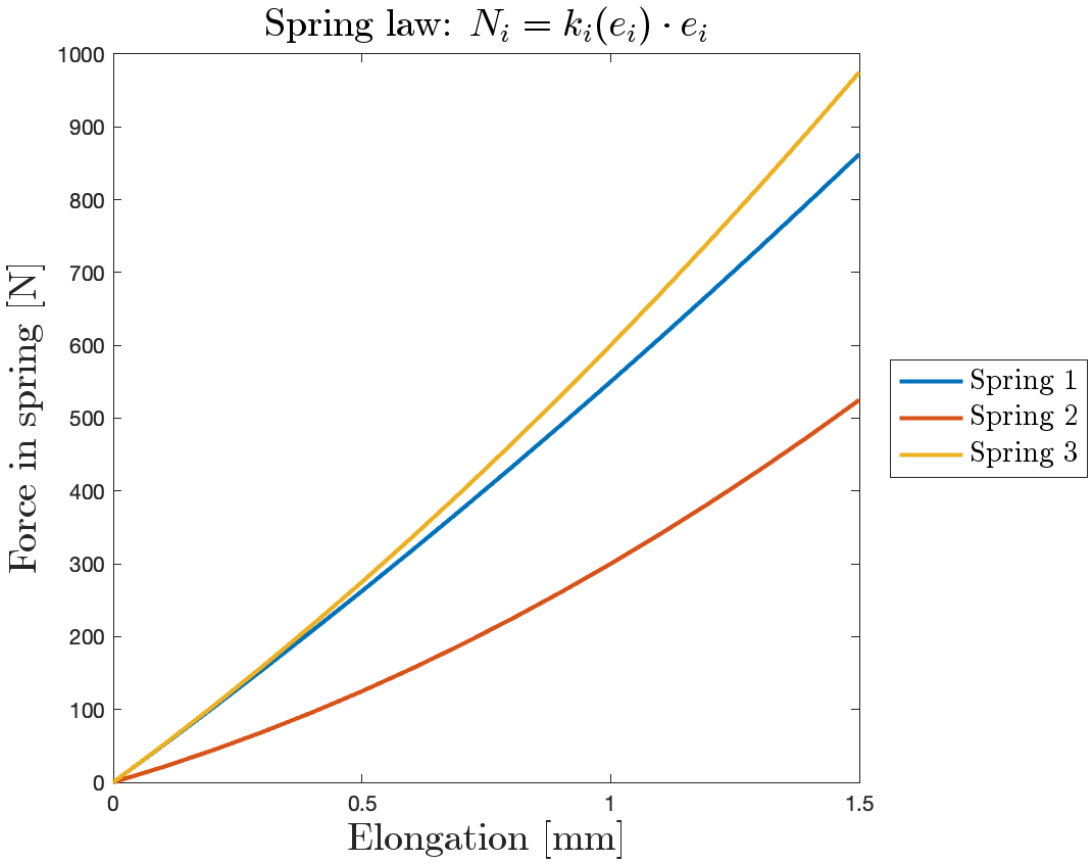


Figure 1.2: Costitutive law plot.

Since springs 1 and 2 are fixed on the wall, their elongation is equivalent to u_1, while for spring 3, the elongation is $u_2 - u_1$. It's very easy to find out the equilibrium of the two nodes:

$$\begin{cases} N_1 + N_2 - N_3 = 0, \\ N_3 = F. \end{cases}$$

It follows that the matrix notation of the system in term of unknown displacement $\mathbf{u}$ is

$$\mathbf{P}(\mathbf{u}) = \mathbf{f} \tag{1.1}$$

or, in a more physical notation,

$$\mathbf{F^{int}}(\mathbf{u}) = \mathbf{F^{ext}},$$

hence, the system is the following:

$$\begin{bmatrix} k_1 + k_2 + k_3 & -k_3 \\ -k_3 & +k_3 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0 \\ F \end{bmatrix}.$$

Note that, as we said before, the above equations are not linear as the stiffness matrix contains unknown variables.

Using the given stiffness of springs, the general matrix equation becomes

$$\begin{cases} (\beta_1 + \beta_2 - \beta_3)\, u_1^2 + (\alpha_1 + \alpha_2 + \alpha_3)\, u_1 + 2\beta_3 u_1 u_2 - \alpha_3 u_2 - \beta_3 u_2^2 = 0 \\ \beta_3 u_1^2 - \alpha_3 u_1 - 2\beta_3 u_1 u_2 + \alpha_3 u_2 + \beta_3 u_2^2 = F \end{cases},$$

which, for the sake of generality, leads to

$$\begin{cases} \mathcal{A} u_1^2 + \mathcal{B} u_1 + \mathcal{C} u_1 u_2 + \mathcal{D} u_2 + \mathcal{E} u_2^2 = 0 \\ \mathcal{F} u_1^2 + \mathcal{G} u_1 + \mathcal{H} u_1 u_2 + \mathcal{I} u_2 + \mathcal{L} u_2^2 = F \end{cases}.$$

In this way it will be easier to build the system in MATLAB, namely with

$$\begin{bmatrix} \mathcal{A} & \mathcal{B} & \mathcal{C} & \mathcal{D} & \mathcal{E} \\ \mathcal{F} & \mathcal{G} & \mathcal{H} & \mathcal{I} & \mathcal{L} \end{bmatrix} \begin{bmatrix} u_1^2 \\ u_1 \\ u_1 u_2 \\ u_2 \\ u_2^2 \end{bmatrix}. \tag{1.2}$$

Finally, the assembled system is

$$\begin{cases} 50u_1^2 + 1200u_1 + 200u_1 u_2 - 500u_2 - 100u_2^2 = 0 \\ 100u_1^2 - 500u_1 - 200u_1 u_2 + 500u_2 + 100u_2^2 = 100 \end{cases}.$$

In the following section, a method of solving the system of nonlinear equations is discussed.

1.3. Numerical Solution

Before discussing systems, it is important to understand the problem of numerical approximation of the zeros of a real-valued function of one variable, that is

$$\boxed{\text{Given } f : (a,b) \to \mathbb{R}, \text{ find } \alpha \in \mathbb{R} \text{ s.t. } f(\alpha) = 0}$$

There are several iterative methods to solve this rootfinding problem, such as the bisection method, the chord method, the secant method, fixed point methods, etc. For a more complete treatment of these, see for example [4].

In the following we analyze another algorithm: the Newton's method, also known as the **Newton–Raphson method**.

Assuming that $f \in \mathcal{C}^1\big((a,b)\big)$ and that α is a simple root of f, if we let

$$q_k = f'(x^k), \qquad \forall\ k \geq 0$$

and assign the initial value x^0, we obtain the Newton scheme

$$x^{k+1} = x^k - \frac{f(x^k)}{q_k}, \qquad k \geq 0. \tag{1.3}$$

The idea is clarified by looking at the first-order series expansion of $f(x)$ in the neighborhood of x^k

$$f(x) \approx f(x^k) + \left[\frac{\mathrm{d}f}{\mathrm{d}x}(x^k)\right](x - x^k) \qquad \xrightarrow{x=x^{k+1}} \qquad \text{Eq. (1.3)}$$

and by Figure 1.3.

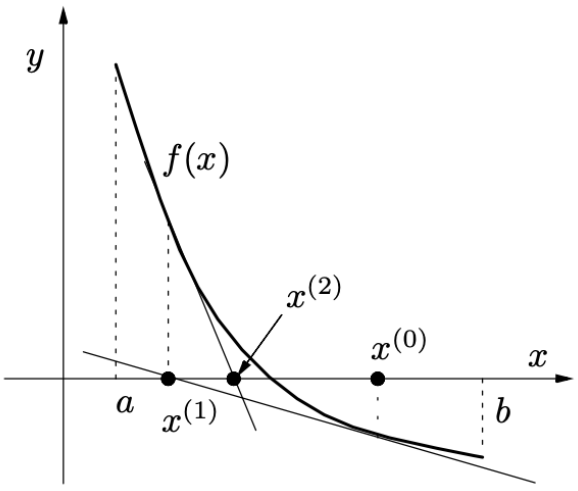


Figure 1.3: Geometric approach of Newton-Raphson method: first two steps.

This method has an high convergence order, namely $p = 2$, i.e.

$$\exists\, C > 0 \qquad \text{s.t.} \qquad \lim_{k\to\infty} \frac{\left|x^{k+1} - \alpha\right|}{\left|x^k - \alpha\right|^2} = C, \tag{1.4}$$

but α should be a simple root and the initial value x^0 should be *close* to the solution, because the method does not always guarantee convergence (actually, sometimes the solution may diverge or oscillate between two points).

An immediate extension of Newton-Raphson's method (1.3) for the nonlinear system (1.1) can be recursively formulated as

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \frac{\mathbf{f} - \mathbf{P}(\mathbf{u}^k)}{\mathbf{K}_T^k} \qquad \text{with} \qquad \mathbf{K}_T^k = \frac{\partial \mathbf{P}}{\partial \mathbf{u}}(\mathbf{u}^k),$$

which leads to the following algorithm:

$$\begin{aligned} &\text{given } \mathbf{u}^0, \text{ for } k = 0, 1, \ldots, \text{ until convergence:} \\ &\text{compute} \quad \mathbf{K}_T^k \text{ and } \mathbf{R}^k = \mathbf{f} - \mathbf{P}(\mathbf{u}^k), && \text{(1.5a)} \\ &\text{solve} \quad \mathbf{K}_T^k\, \Delta\mathbf{u}^k = \mathbf{R}^k, && \text{(1.5b)} \\ &\text{set} \quad \mathbf{u}^{k+1} = \mathbf{u}^k + \Delta\mathbf{u}^k. && \text{(1.5c)} \end{aligned}$$

In general, this solution will not satisfy the system of nonlinear equations exactly and there will be some residual (or unbalance force in a physical point of view). If the k-th residual is smaller than a given tolerance, the solution $\mathbf{u}^{k+1}$ can be accepted as the accurate solution, and the process stops. The stopping criterion is expressed as follows

$$\text{conv}^{k+1} = \frac{\displaystyle\sum_{j=1}^{n} \left(\mathrm{R}_j^{k+1}\right)^2}{1 + \displaystyle\sum_{j=1}^{n} \left(\mathrm{f}_j\right)^2}, \tag{1.6}$$

and, in order to show quadratic convergence as in Eq. (1.4), it is often enough to show that **conv** reduces quadratically at each iteration.

! From Eq. (1.5b), one can notice that at each step k the solution of a linear system with matrix $\mathbf{K}_T^k$ is required, thus the matrix cannot be singular. We will use the *backslash* MATLAB command to solve the system, but the computational cost increases as n^2. There are several ways to solve this, for example use the LU decomposition and keep it fixed for some iterations, or just keep $\mathbf{K}_T^k$ fixed. See Secion 1.4 for further details.

1.4. MATLAB Implementation

Here we present and explain the MATLAB scripts used to solve the problem. Algorithms are listed in the Subsection 1.4.1.

In `main11` (Algorithm 1.1) we implement the Newton-Raphson method previously described. Here two auxiliary functions are used to compute the internal force vector $\mathbf{P}(\mathbf{u})$ and its Jacobian matrix $\mathbf{K}_T$ at each iteration. These MATLAB functions `generate_int_force` (Algorithm 1.2) and `generate_jacobian` (Algorithm 1.3) are merely based on Eq. (1.2).

The output of the code is the following:

```
**********************************************************************
RESULTS
**********************************************************************

iter   u1        u2        conv
   0   2.00000   2.00000   4.00960e+02
   1   0.53846   0.73846   1.00124e+01
   2   0.16655   0.35914   4.30187e-02
   3   0.13889   0.33147   1.31776e-06
   4   0.13873   0.33132   1.29140e-15

Convergence reached in 4 iterations

**********************************************************************
```

The initial guess choosen was $\mathbf{u}^0 = [2; 2]$, but we can easily repeat the process for other initial estimates. For example, with $\mathbf{u}^0 = [0; 0]$ we have

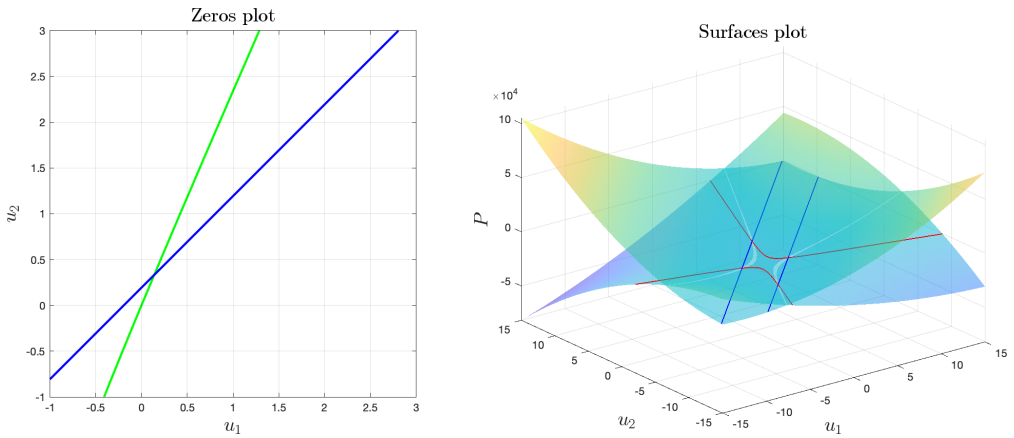
```
**********************************************************************
RESULTS
**********************************************************************

iter   u1        u2        conv
   0   0.00000   0.00000   9.99900e-01
   1   0.14286   0.34286   1.68796e-03
   2   0.13874   0.33133   3.87449e-09
   3   0.13873   0.33132   1.81925e-20

Convergence reached in 3 iterations

**********************************************************************
```

and with $\mathbf{u}^0 = [9; 9]$ we have

```
**********************************************************************
RESULTS
**********************************************************************

iter    u1          u2          conv
   0    9.00000     9.00000     3.40378e+04
   1    3.60294     3.80294     1.90534e+03
   2    1.14953     1.34212     8.15108e+01
   3    0.28541     0.47799     1.25440e+00
   4    0.14284     0.33542     9.29483e-04
   5    0.13874     0.33132     6.38409e-10

Convergence reached in 5 iterations

**********************************************************************
```

From all these outputs we can say that as MATLAB program iterates, the displacements converges to the numerical solution $\mathbf{u} = [0.13873; 0.33132]$ mm; secondly, the number of iterations depends on the starting point, however the residual reduction via convergence criterion (1.6) is approximately quadratic.

We can double check the accuracy of our solution via plotting the surfaces of the system of nonlinear equations with zero-level contour. The results we obtain with `main12` (Algorithm 1.4) ant the `plotzeros` function (Algorithm 1.5) are the following figures:



(a) Zero-level contour plot. (b) 3D Surface plot.

Figure 1.4: Zero-level contour and Surface plots.

Note that there exist four solutions to the problem, but only one of them, the one we found, is positive.

Thanks to script `main13` (Algorithm 1.6) we generate useful data for the subsequent processing in Abaqus (see Section 1.5) and Figure 1.2.

The Newton–Raphson method requires that at each iteration, the Jacobian matrix should be formed and the system of linearized equations should be solved for the increment of the solution. Computationally, these are expensive tasks. The **Modified Newton–Raphson method** is an attempt to make these procedures less expensive. Instead of formulating a new tangent stiffness matrix at each iteration, the initial tangent stiffness matrix is repeatedly used for all iterations.

Through `main14` (Algorithm 1.7) we implement the Modified Newton-Raphson method, gaining the following output:

```
**********************************************************************
RESULTS
**********************************************************************

iter   u1        u2        conv
   0   2.00000   2.00000   4.00960e+02
   1   0.53846   0.73846   1.00124e+01
   2   0.29199   0.48399   1.38037e+00
   3   0.20185   0.39448   2.24486e-01
   4   0.16538   0.35796   3.94896e-02
   5   0.15010   0.34268   7.13582e-03
   6   0.14360   0.33618   1.30509e-03
   7   0.14082   0.33340   2.39875e-04
   8   0.13963   0.33221   4.41837e-05
   9   0.13912   0.33170   8.14585e-06
  10   0.13890   0.33148   1.50238e-06
  11   0.13880   0.33139   2.77140e-07
  12   0.13876   0.33135   5.11268e-08
  13   0.13875   0.33133   9.43218e-09
  14   0.13874   0.33132   1.74013e-09
  15   0.13874   0.33132   3.21035e-10

Convergence reached in 15 iterations

**********************************************************************
```

The method usually requires a greater number of iterations for convergence than that of the regular Newton–Raphson method. However, the overall computational cost to obtain the solution can be made less because each iteration is much faster than that of the regular Newton–Raphson method. The method is also a little more stable and is not prone to divergence.

1.4.1. MATLAB scripts

Algorithm 1.1: main11

```
clear; clc; close;
% data
tol = 1.0e-9; iter = 0; max_iter = 100; u = [2;2]; old_u = u; F = 100;
a1 = 500; b1 = 50; a2 = 200; b2 = 100; a3 = 500; b3 = 100;
k1 = a1+b1*u(1); k2 = a2+b2*u(1); k3 = a3+b3*(u(2)-u(1));
% internal force vector
P = generate_int_force(u,a1,b1,a2,b2,a3,b3);
% external force vector
FF = [0; F];
% residual
R = FF - P;
% convergence parameter
conv = (R(1)^2+R(2)^2)/(1+FF(1)^2+FF(2)^2);
conv_rate = [];
fprintf('%s', repmat('*', 1, 69));
fprintf('\nRESULTS\n');
fprintf('%s', repmat('*', 1, 69));
fprintf('\n');
fprintf('\niter   u1        u2        conv');
fprintf('\n %3d   %7.5f   %7.5f   %7.5e',iter,u(1),u(2),conv);
while conv > tol && iter < max_iter
    % jacobian matrix of P
    Kt = generate_jacobian(u,a1,b1,a2,b2,a3,b3);
    % check if it is singular
    if det(Kt)==0
        fprintf('\n\n');
        error('Jacobian is singular!')
    end
    % solve the linear system
    delta_u = Kt\R;
    % update of the solution
    u = old_u + delta_u;
    % update of the internal force vector
    P = generate_int_force(u,a1,b1,a2,b2,a3,b3);
    % update of the residual
    R = FF - P;
    % update of the convergence parameter
    conv = (R(1)^2+R(2)^2)/(1+FF(1)^2+FF(2)^2);
    % save it in a vector
    conv_rate = [conv_rate; conv];
    % ready for a new iteration
    old_u = u;
```

```
    iter = iter + 1;
    % print of the results
    fprintf('\n %3d   %7.5f   %7.5f   %7.5e',iter,u(1),u(2),conv);
end
if iter==max_iter
    fprintf('\n\n');
    error('Convergence not reached!')
end
fprintf('\n\nConvergence reached in %d iterations',iter);
fprintf('\n\n');
fprintf('%s', repmat('*', 1, 69)); fprintf('\n\n');
% p = log2(conv_rate(1:end-1)./conv_rate(2:end));
```

Algorithm 1.2: generate int force

```
function P = generate_int_force(u,a1,b1,a2,b2,a3,b3)
A = b1+b2-b3; B = a1+a2+a3; C = 2*b3; D = -a3; E = -b3;
F = b3; G = -a3; H = -2*b3; I = a3; L = b3;
U = [u(1)^2;u(1);u(1)*u(2);u(2);u(2)^2];
P1 = [A,B,C,D,E;F,G,H,I,L];
P = P1*U;
end
```

Algorithm 1.3: generate jacobian

```
function J = generate_jacobian(u,a1,b1,a2,b2,a3,b3)
A = b1+b2-b3; B = a1+a2+a3; C = 2*b3; D = -a3; E = -b3;
F = b3; G = -a3; H = -2*b3; I = a3; L = b3;
U = [u(1);u(2);1];
J1 = [2*A,C,B;C,2*E,D;2*F,H,G;H,2*L,I];
J2 = J1*U;
J = [J2(1),J2(2);J2(3),J2(4)];
end
```

Algorithm 1.4: main12

```
clear; clc; close;
a1 = 500; b1 = 50; a2 = 200; b2 = 100; a3 = 500; b3 = 100; f = 100;
A = b1+b2-b3; B = a1+a2+a3; C = 2*b3; D = -a3; E = -b3;
F = b3; G = -a3; H = -2*b3; I = a3; L = b3;
fun = @(u) [A*u(1)^2+B*u(1)+C*u(1)*u(2)+D*u(2)+E*u(2)^2;
            F*u(1)^2+G*u(1)+H*u(1)*u(2)+I*u(2)+L*u(2)^2-f];
plotzeros(fun, [-1 3], [-1 3])
%
figure()
[X,Y] = meshgrid(-15:.5:15);
Z1 = (b1+b2-b3)*X.^2+(a1+a2+a3)*X+2*b3*X.*Y-a3*Y-b3*Y.^2;
s1 = surf(X,Y,Z1,'FaceAlpha',0.5);
s1.EdgeColor = 'none';
xlabel('$u_1$','Interpreter','latex',FontSize=18)
ylabel('$u_2$','Interpreter','latex',FontSize=18)
zlabel('$P$','Interpreter','latex',FontSize=18)
title('Surfaces plot','Interpreter','latex',FontSize=18)
hold on
Z2 = b3*X.^2-a3*X-2*b3*X.*Y+a3*Y+b3*Y.^2-F;
s2 = surf(X,Y,Z2,'FaceAlpha',0.5);
s2.EdgeColor = 'none';
contour(X,Y,Z1, [0 0], 'r', 'LineWidth', 1);
contour(X,Y,Z2, [0 0], 'b', 'LineWidth', 1);
```

Algorithm 1.5: plotzeros

```
function plotzeros(fun, x_limits, y_limits, varargin)
if nargin > 3
    sample_len = varargin{1};
else
    sample_len = 500;
end
v_dis_x = linspace(x_limits(1), x_limits(2), sample_len);
v_dis_y = linspace(y_limits(1), y_limits(2), sample_len);
n_dis_x = length(v_dis_x);
n_dis_y = length(v_dis_y);
Z_1 = zeros(n_dis_y, n_dis_x);
Z_2 = zeros(n_dis_y, n_dis_x);
for i = 1:n_dis_x
    for j = 1:n_dis_y
        funz = fun([v_dis_x(i); v_dis_y(j)]);
        Z_1(j,i) = funz(1); Z_2(j,i) = funz(2);
    end
end
```

```
contour(v_dis_x, v_dis_y, Z_1, [0 0], 'Linecolor',[0 1 0], 'Linewidth',
   2)
hold on
contour(v_dis_x, v_dis_y, Z_2, [0 0], 'Linecolor',[0 0 1], 'Linewidth',
   2)
axis equal
grid on
xlabel('$u_1$','Interpreter','latex',FontSize=18)
ylabel('$u_2$','Interpreter','latex',FontSize=18)
title('Zeros plot','Interpreter','latex',FontSize=18)
end
```

Algorithm 1.6: main13

```
clear; clc; close; format shorte
U = 0:0.1:1.5; Ut = U'
a1 = 500; b1 = 50; a2 = 200; b2 = 100; a3 = 500; b3 = 100;
k1 = a1+b1*Ut; k2 = a2+b2*Ut; k3 = a3+b3*Ut; kEQ = k1+k2;
F1 = k1.*Ut;
figure(1)
plot(Ut,F1,'linewidth',2)
xlabel('Elongation [mm]','Interpreter','latex',FontSize=18)
ylabel('Force in spring [N]','Interpreter','latex',FontSize=18)
title('Spring law: $N_i=k_i(e_i)\cdot e_i$','Interpreter','latex',
   FontSize=18)
hold on
F2 = k2.*Ut;
plot(Ut,F2,'linewidth',2)
F3 = k3.*Ut;
plot(Ut,F3,'linewidth',2)
legend('Spring 1','Spring 2','Spring 3','Interpreter','latex',Location='
   eastoutside',FontSize=14)
F12 = kEQ.*Ut
figure(2)
plot(Ut,F12,'linewidth',2)
xlabel('Elongation [mm]','Interpreter','latex',FontSize=18)
ylabel('Force in spring [N]','Interpreter','latex',FontSize=18)
title('Spring law: $N_i=k_i(e_i)\cdot e_i$','Interpreter','latex',
   FontSize=18)
hold on
F3 = k3.*Ut
plot(Ut,F3,'linewidth',2)
legend('Spring 1+2','Spring 3','Interpreter','latex',Location='
   eastoutside',FontSize=14)
```

Algorithm 1.7: main14

```
clear; clc; close;
tol = 1.0e-9; iter = 0; max_iter = 100; u = [2;2]; old_u = u; F = 100;
a1 = 500; b1 = 50; a2 = 200; b2 = 100; a3 = 500; b3 = 100;
k1 = a1+b1*u(1); k2 = a2+b2*u(1); k3 = a3+b3*(u(2)-u(1));
P = generate_int_force(u,a1,b1,a2,b2,a3,b3);
% jacobian matrix of P: FIXED
Kt = generate_jacobian(u,a1,b1,a2,b2,a3,b3);
FF = [0; F];
R = FF - P;
conv = (R(1)^2+R(2)^2)/(1+FF(1)^2+FF(2)^2);
conv_rate = [];
fprintf('%s', repmat('*', 1, 69));
fprintf('\nRESULTS\n');
fprintf('%s', repmat('*', 1, 69));
fprintf('\n');
fprintf('\niter   u1        u2        conv');
fprintf('\n %3d   %7.5f   %7.5f   %7.5e',iter,u(1),u(2),conv);
while conv > tol && iter < max_iter
    delta_u = Kt\R;
    u = old_u + delta_u;
    P = generate_int_force(u,a1,b1,a2,b2,a3,b3);
    R = FF - P;
    conv = (R(1)^2+R(2)^2)/(1+FF(1)^2+FF(2)^2);
    conv_rate = [conv_rate; conv];
    old_u = u;
    iter = iter + 1;
    fprintf('\n %3d   %7.5f   %7.5f   %7.5e',iter,u(1),u(2),conv);
end
if iter==max_iter
    fprintf('\n\n');
    error('Convergence not reached!')
end
fprintf('\n\nConvergence reached in %d iterations',iter);
fprintf('\n\n');
fprintf('%s', repmat('*', 1, 69)); fprintf('\n\n');
% p = log2(conv_rate(1:end-1)./conv_rate(2:end));
```

1.5. Abaqus Implementation

Here we solve the problem through the Abaqus Software by using spring-like connectors. The adopted units of measure are mm and N in order to be consistent, as Abaqus requires.

Before starting, let's remark that our nonlinear system shown in Figure 1.1 can be semplified in a system of two serial connected springs:

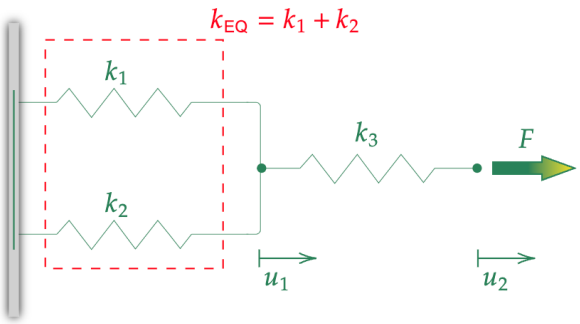


Figure 1.5: Equivalent system of nonlinear springs.

1.5.1. Interaction module

We define the nodes of the spring system by setting three colinear reference points (the distance between the nodes is not important) using the "Create Reference Point" button. Then we re-scale the view and the result is shown in Figure 1.6.

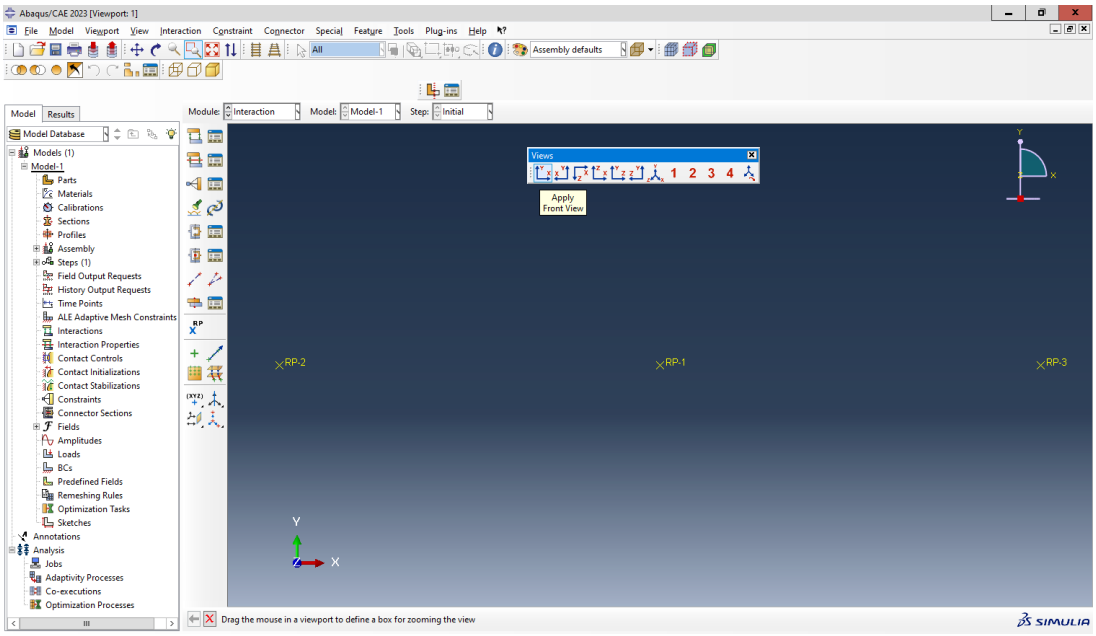


Figure 1.6: Reference points.

We proceed by creating the wire feature by clicking on the "Create Wire Feature"

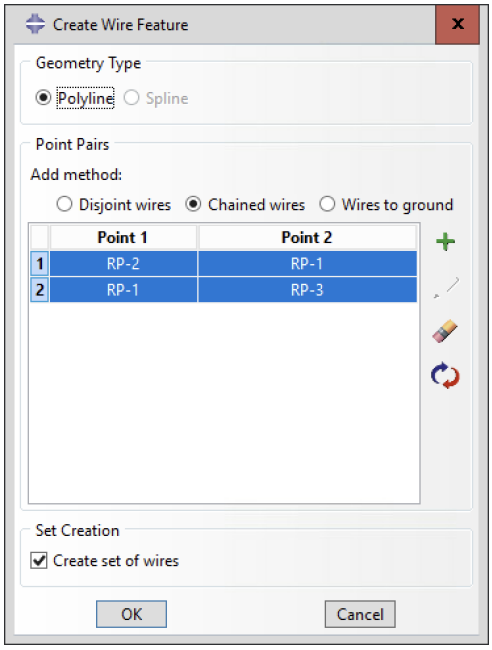


Figure 1.7: Wire features.

and the result is the following:

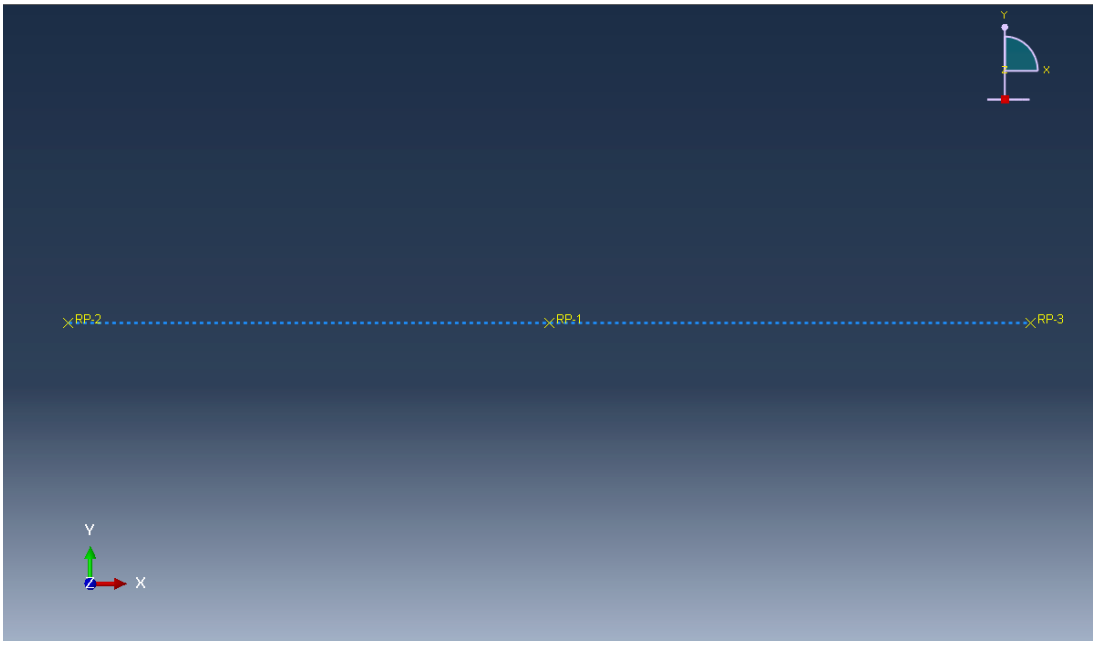


Figure 1.8: Middle result of Interaction module.

A new *section* should be defined for each springs. In the "Connector Section Manager", for both springs, we set the connection category as basic and the connection translational type as axial:

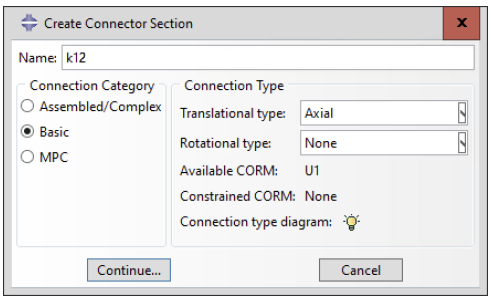
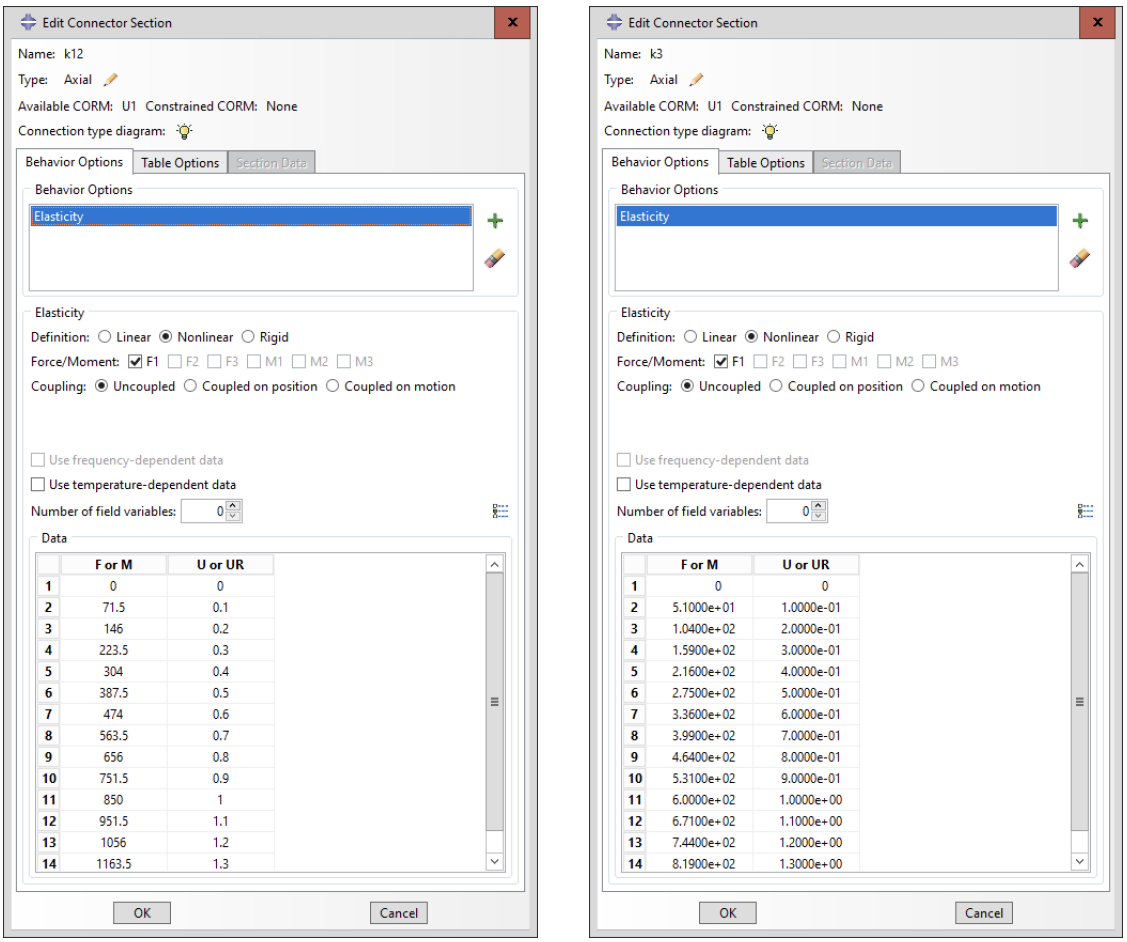


Figure 1.9: Connector Section features.

Here we manage to introduce nonlinearity by pasting the tables of values obtained with Algorithm 1.6 in MATLAB:



(a) Behavior options of Spring 1+2.

(b) Behavior options of Spring 3.

Figure 1.10: Behavior options of the springs.

Finally, we associate each spring with its respective mechanical property:

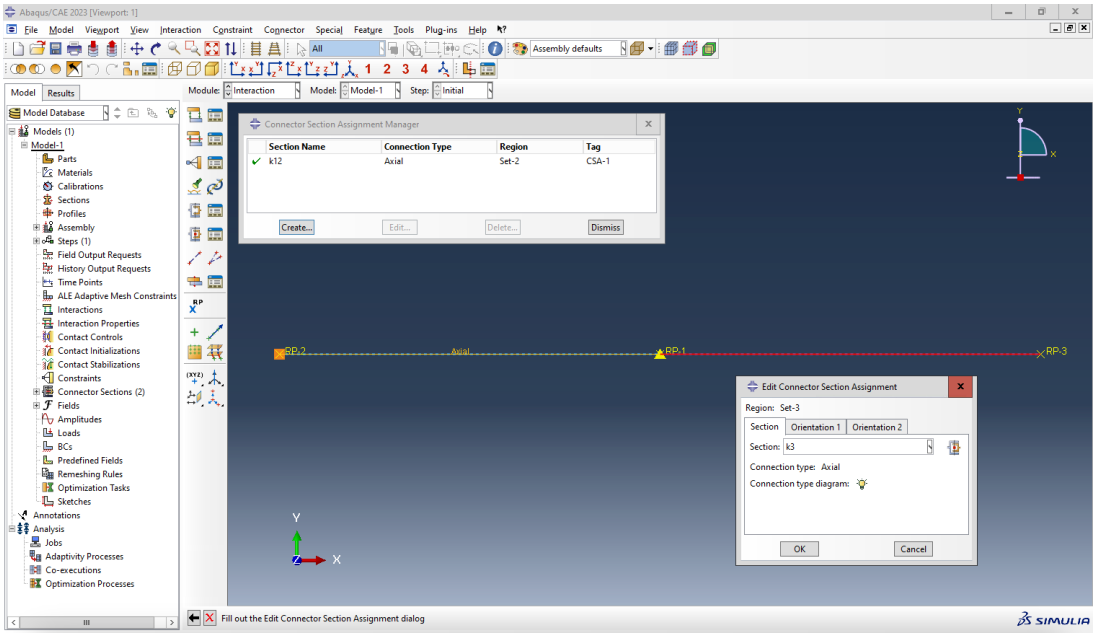


Figure 1.11: Connector Section Assignment.

The final result of this module is shown in Figure 1.12.

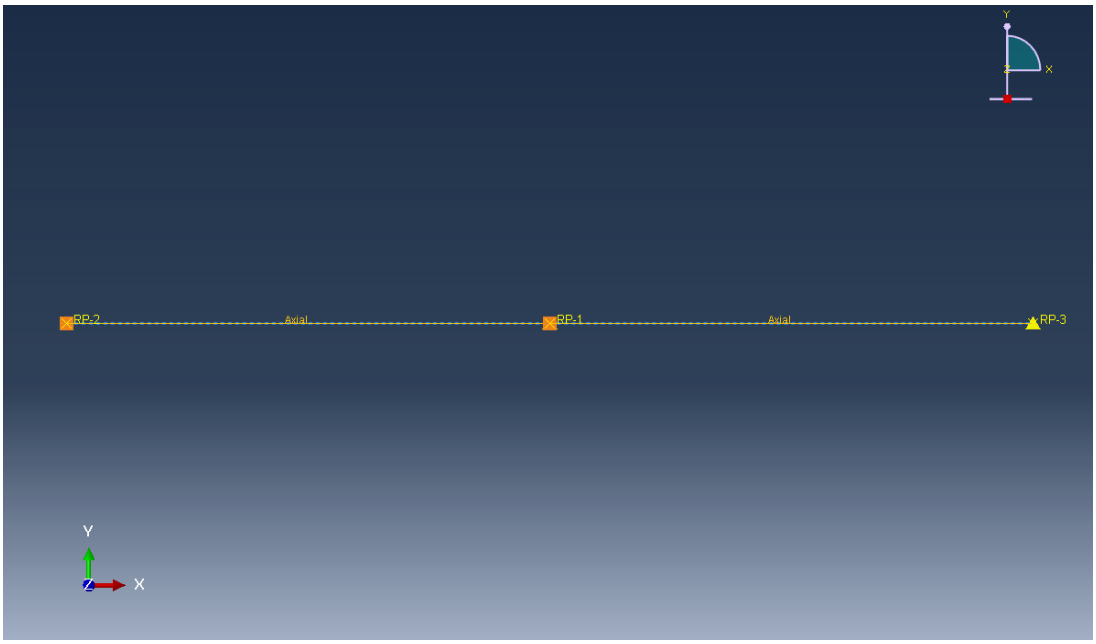


Figure 1.12: Final result of Interaction module.

1.5.2. Step module

We click on the "Step Manager" button, which looks like Figure 1.13,

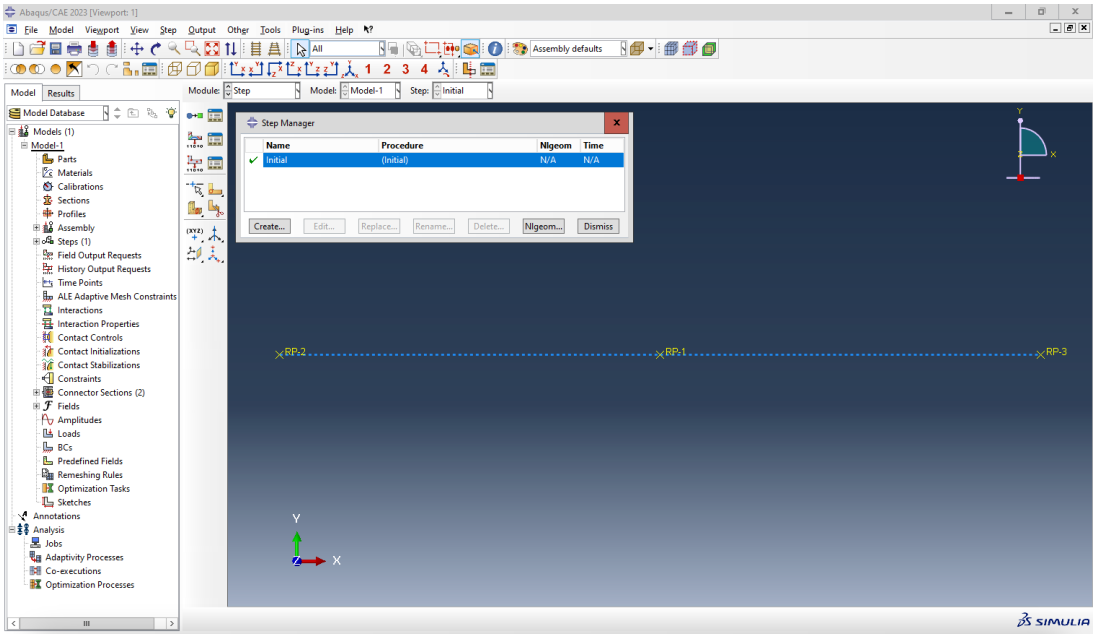


Figure 1.13: Step Manager.

and we create a new step turning on the Nlgeom option (in order to include the nonlinear effects of large displacements):

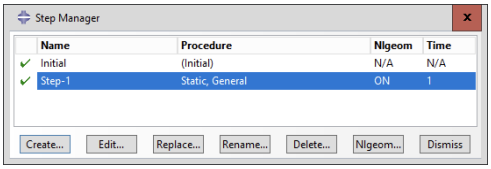


Figure 1.14: Final result of Step module.

1.5.3. Load module

In this module we first set the boundary conditions of the problem. For the node on the left we want to restrict the vertical, horizontal, and the out-of-plane displacements. Thus, the node on the left has U1, U2 and U3 restricted. Instead, the middle and the right nodes have just U2 and U3 restricted. Figure 1.15 shows the setting.

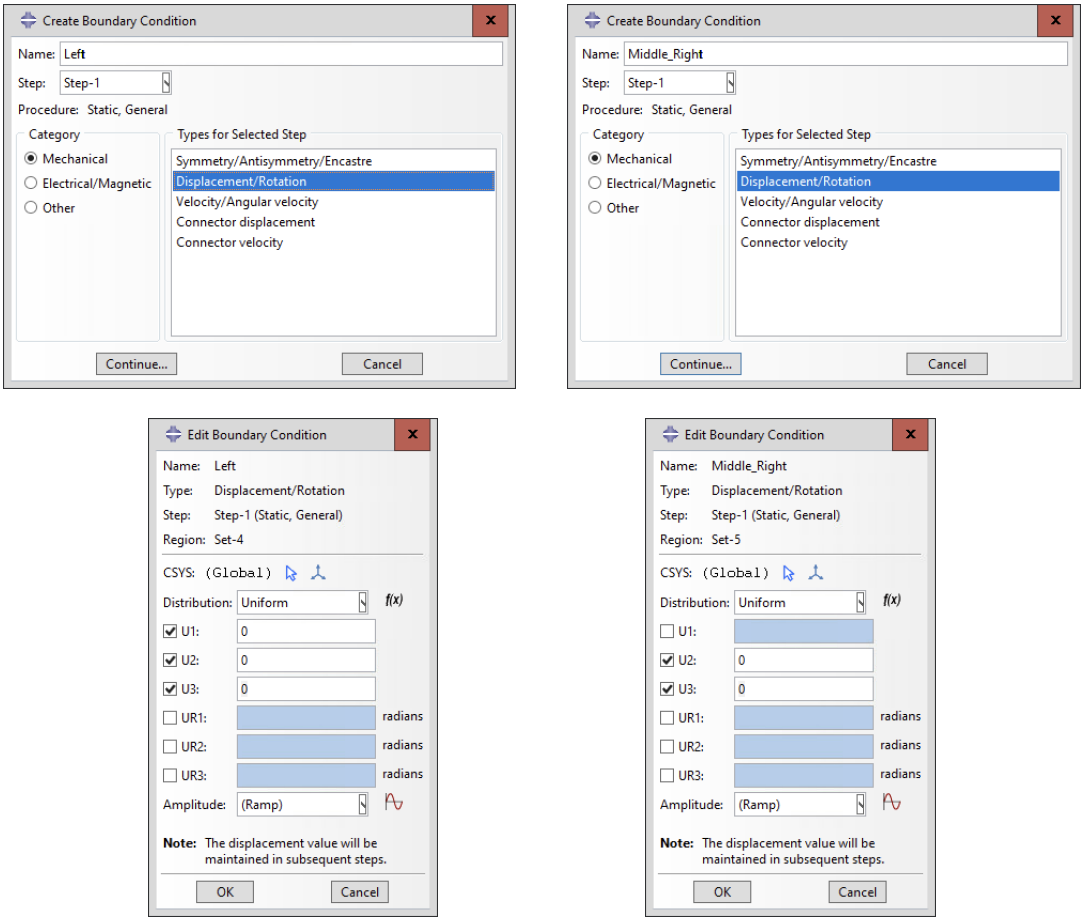


Figure 1.15: Boundary conditions of the problem.

Then we set a concentrated force of 100 N acting on the right node in the x direction with the "Create Load" symbol:

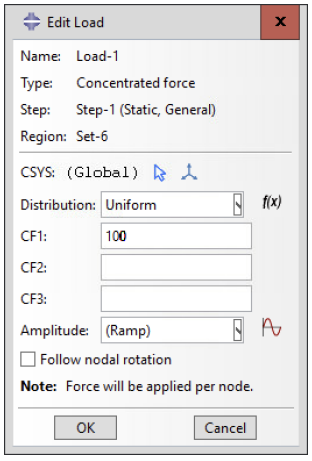


Figure 1.16: Load options.

Here there is the result:

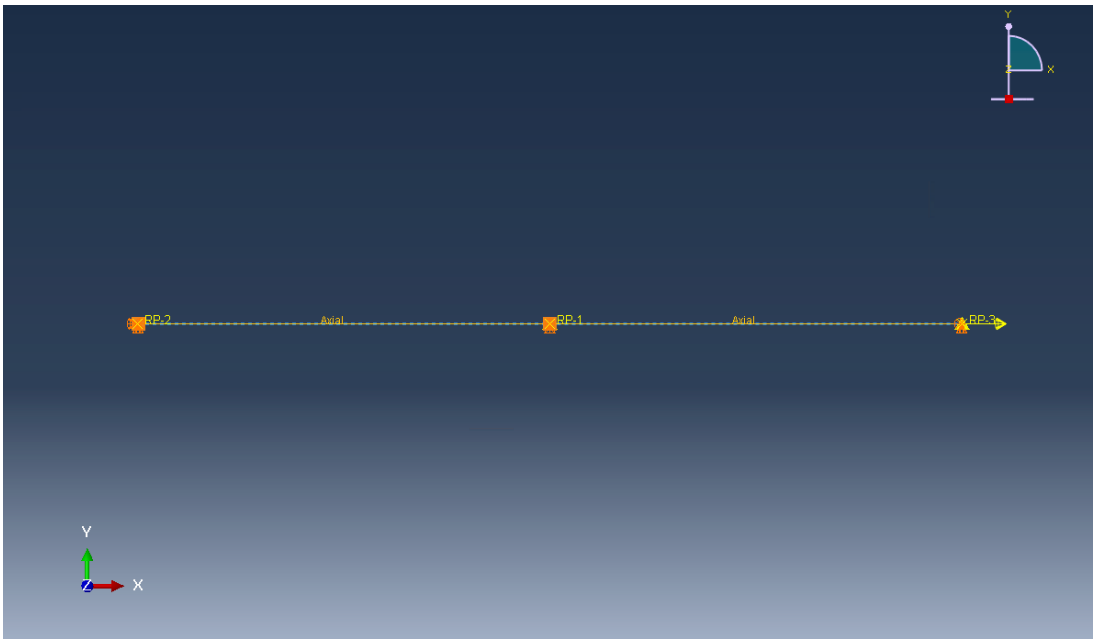


Figure 1.17: Final result of Load module.

1.5.4. Job module

We create a standard job in the "Job Manager" and we submit it:

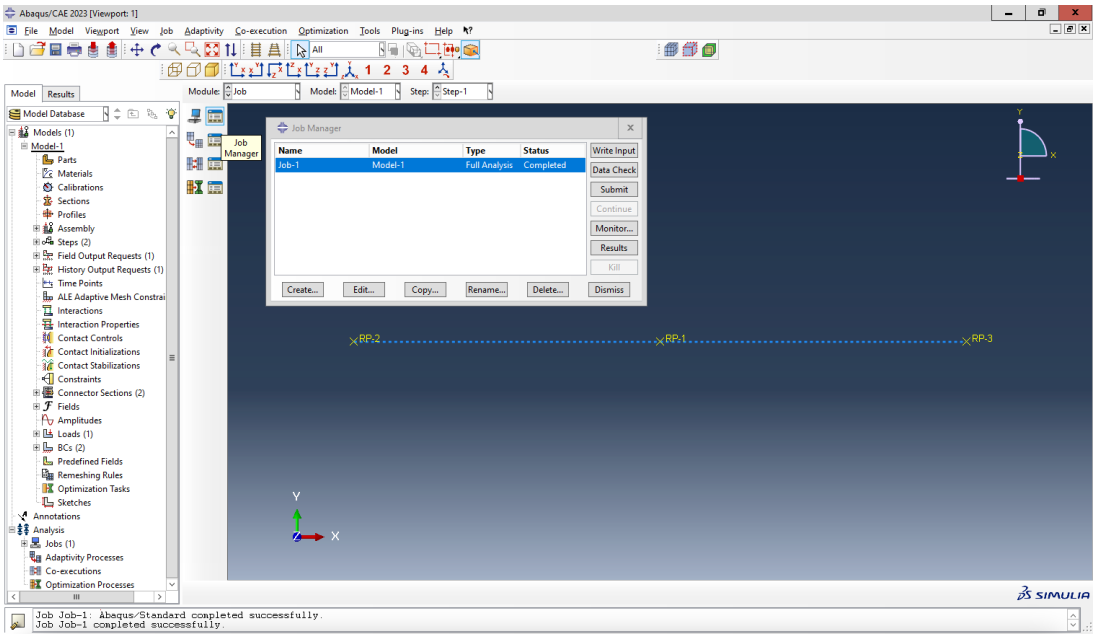


Figure 1.18: Final result of Job module.

1.5.5. Visualization module

First, we enable the connectors view in the ODB Display Options, as in Figure 1.19

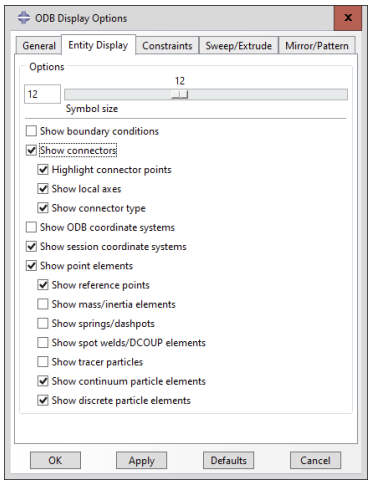


Figure 1.19: View ODB Display Options.

Here is the result we obtained:

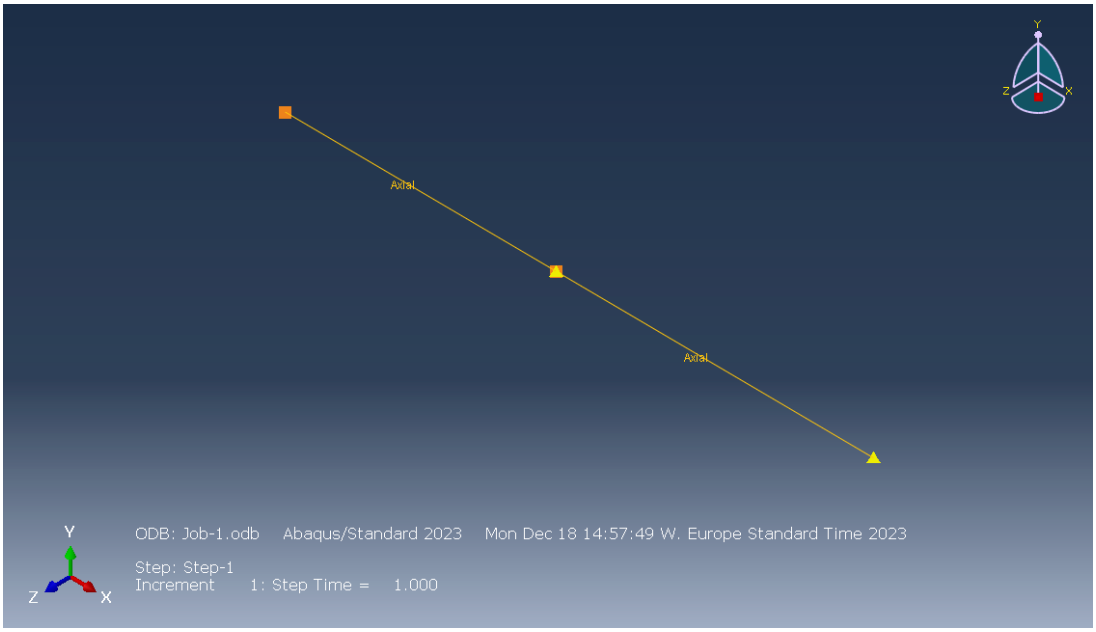


Figure 1.20: Final result.

We can explain U1 thanks to Figure 1.21.

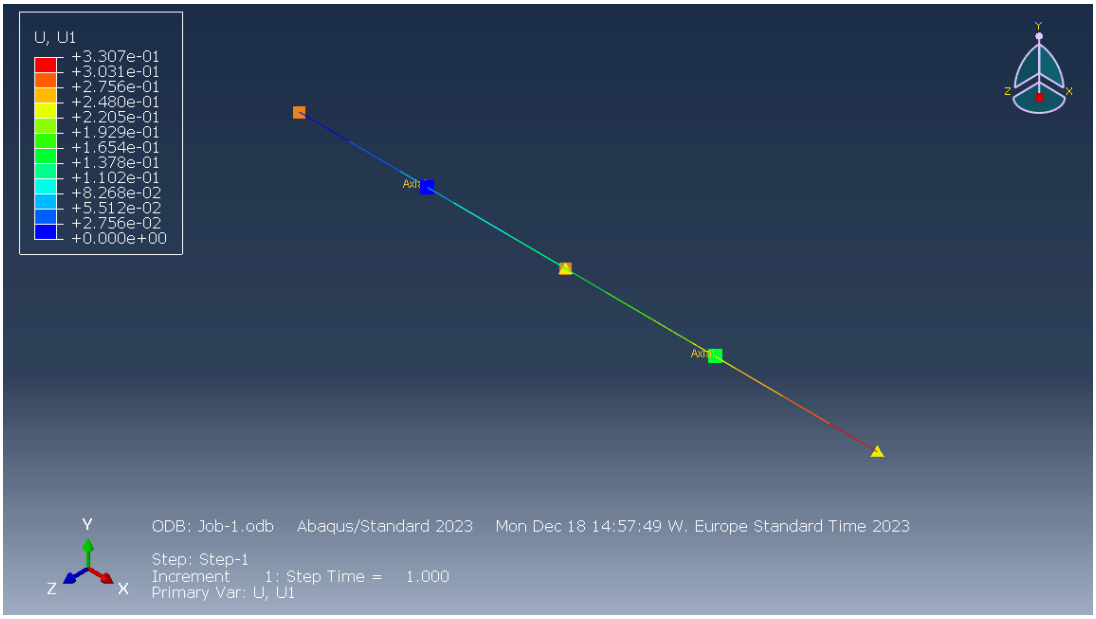


Figure 1.21: Final result in term of U1.

To be more precise, we proceed by clicking on the "Create XY Data" symbol and then we select the spatial displacement of U1 of the middle and right nodes (the ones we are interested in):

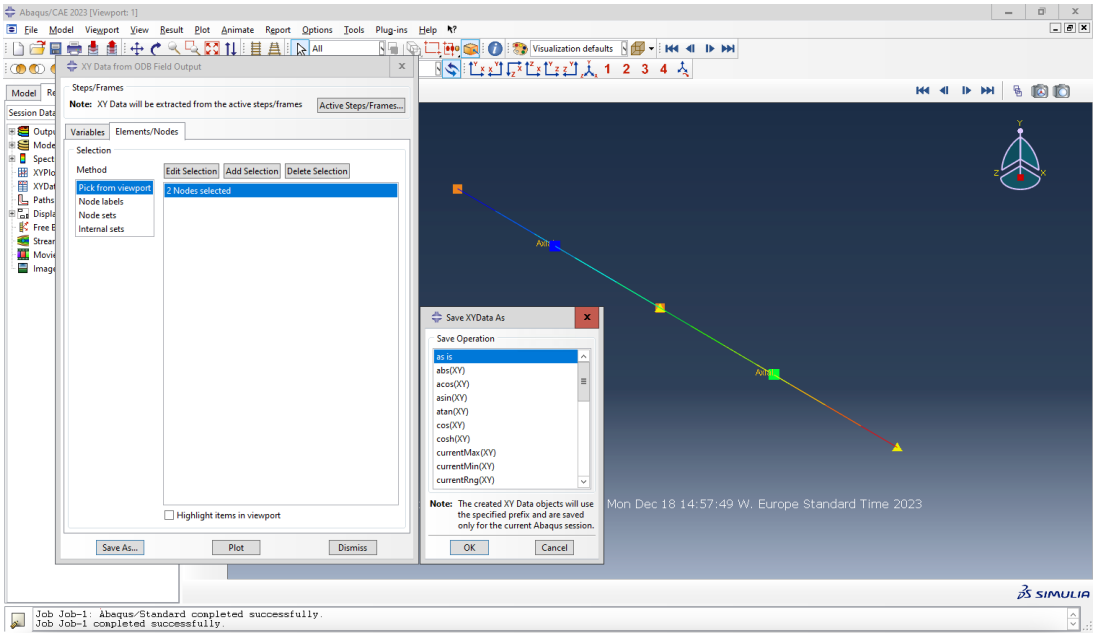


Figure 1.22: XY Data from ODB Field Output.

Finally, by going into the *edit* option of each displacement we can check the numerical result for each selected node:

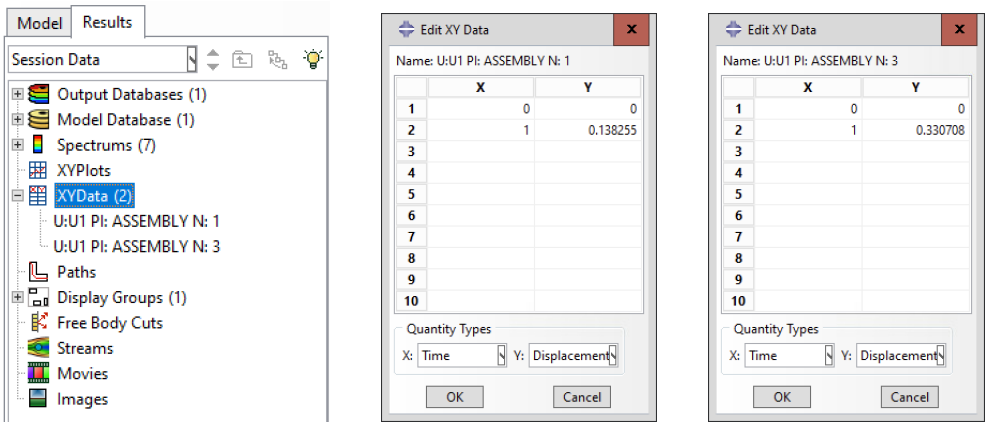


Figure 1.23: Numerical results with Abaqus.

1.6. Comparison between MATLAB and Abaqus

Now we compare the values previously obtained with MATLAB and Abaqus.

Quantity	Symbol	MATLAB	Abaqus	UoM
Horizontal middle displacement	u_1	0.13873	0.138255	mm
Horizontal right displacement	u_2	0.33132	0.330708	mm

Table 1.2: MATLAB and Abaqus results.

In conclusion, the analysis of the results obtained through simulation in MATLAB and Abaqus confirms the accuracy of the proposed model, highlighting a congruent agreement between the two software and the adopted theoretical methodology.

2 | Potential Problems with BEM

2.1. Introduction

The past fifty years have been marked by the evolution of computers and an enormous availability of computational power. This has boosted the development of computational methods and their application in engineering. The most popular computational method is the finite element method (FEM), however around 1970 the engineering community started to develop the boundary element method (BEM).

The term *Boundary Element Method* was coined in 1977 in three publications: Banerjee and Butterfield, Brebbia and Dominguez, and Dominguez. The mathematical foundations were established by Betti in 1872 and Somigliana in 1886 for elasticity problems, while Green in 1828 and Fredholm in 1903 made foundational contributions to potential problems.

A fair question to ask is "why do we need the BEM since we already have the FEM that solves engineering problems?". The answer is that modeling with finite elements can be ineffective and laborious for certain classes of problems. So the FEM, despite the generality of its application in engineering problems, is not free of drawbacks. The most important advantages of preferring BEM over FEM are:

1. The solution is mathematically expressed as a continuous mathematical formula, namely a Boundary Integral Equation (BIE). Therefore the associated numerical method, the BEM, benefits of a discretization only over the boundary Γ of the problem domain Ω, leading to a significant reduction in the number of degrees of freedom of the numerical model (see Fig 2.1).

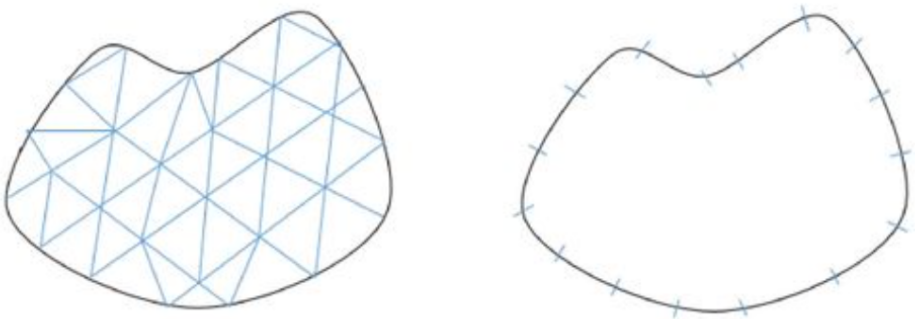


Figure 2.1: FEM versus BEM domain discretization.

2. The method is particularly effective in computing accurately the derivatives of the field function (e.g., fluxes, strain, stresses, moments). Instead, in FE methods the accuracy drops considerably in areas of large gradients.

   However, we mention that in mixed Neumann Dirichlet problems with Lipschitz domain there arises an issue when considering continuous elements. A possible solution is the so-called double node technique, where, thanks to the splitting of a physical node into more computational nodes, continuity is preserved only on the solution, while its normal gradient is allowed to have a jump across physical edges.

3. For infinite domains, the problem is formulated simply as an exterior one. In this manner computer programs developed for finite domains can be used, with just a few modifications, to solve problems in infinite domains. This is not possible with the FEM.

4. The method is well suited for solving problems in domains with geometric peculiarities, such as cracks. Moreover, BEM is more feasible for problems described by differential equations of fourth or higher order (e.g., plate equation).

On the other hand, the BEM exhibits the following inherent main disadvantages:

6. Application of the BEM requires the establishment of the BIE. This is possible only if the problem is linear and its fundamental solution can be established, such as for Laplace equation, Helmholtz equation, and Stokes system. Hence, the method cannot be used for problems whose fundamental solution is either unknown or cannot be determined. Such are, for example, problems described by differential equations with variable coefficients.

7. The numerical implementation of the BEM results in systems of linear algebraic equations whose coefficient matrices are fully populated and nonsymmetrical. Moreover, if one considers mixed Neumann Dirichlet boundary value problems the final linear system is generally ill conditioned. In a FEM model, however, the corresponding matrices have some very nice properties, they are banded and symmetric. This drawback of the BEM is counterbalanced by a smaller dimension of its matrices (see Figure 2.2).

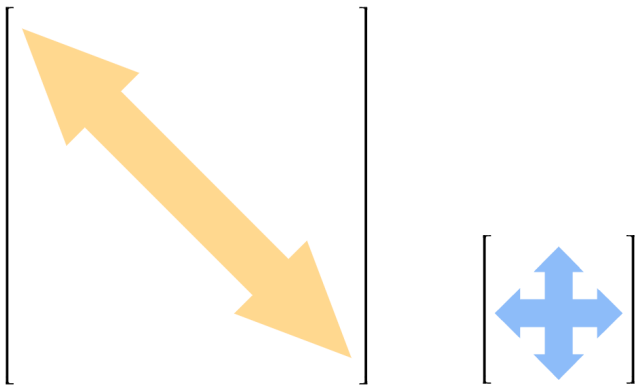


Figure 2.2: FEM versus BEM coefficient matrix.

Nowadays, boundary integral formulation have been applied to problems involving hydrodynamic flows (e.g., Figure 2.3), flow around aerodynamic lifting bodies, structural mechanics, electrostatics, quantum mechanics, and acoustics (e.g., Figure 2.4). See [3] for more delailed explanations.

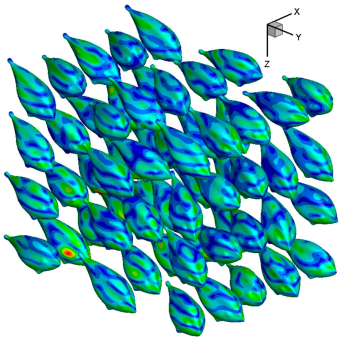


Figure 2.3: Scattering of a multiple fish model.

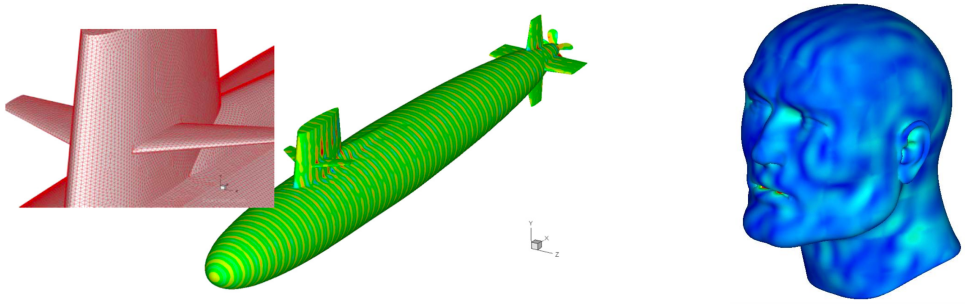


Figure 2.4: (Left) BEM model of the Skipjack submarine impinged upon by an incident wave in the direction $(1, 0, -1)$. (Right) BEM mesh and sound-pressure plots for a human-head model.

In this chapter we are going to present the theoretical background and several numerical application of the BEM, mainly referring to [1].

2.2. Boundary Integral Equations

We consider a bounded open domain $\Omega \subseteq \mathbb{R}^2$ with Lipschitz boundary $\Gamma = \partial\Omega$, and we want to solve engineering problems described by the potential equation

$$\Delta u = f(x, y) \qquad \text{in } \Omega. \tag{2.1}$$

This is the Poisson equation, which for $f = 0$ is known as the Laplace equation. Here $u : \mathbb{R}^2 \to \mathbb{R}$ is the potential produced at a point (x, y) in the domain Ω due to a source $f(x, y)$. Usually, Eq. (2.1) is coupled with the following boundary conditions:

$$\text{Dirichlet b.c.} \qquad u = \overline{u} \qquad \text{on } \Gamma \tag{2.2}$$

$$\text{Neumann b.c.} \qquad \frac{\partial u}{\partial n} = \overline{u}_n \qquad \text{on } \Gamma \tag{2.3}$$

$$\text{Mixed b.c.} \qquad \begin{aligned} u &= \overline{u} && \text{on } \Gamma_1 \\ \frac{\partial u}{\partial n} &= \overline{u}_n && \text{on } \Gamma_2 \end{aligned} \tag{2.4}$$

where $\dfrac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n}$, $\Gamma_1 \cup \Gamma_2 = \Gamma$ and $\Gamma_1 \cap \Gamma_2 = \varnothing$.

2.2.1. Fundamental Solution

Before dealing with a bounded domain, let us solve the Poisson equation in the whole $\mathbb{R}^2$. As phisics suggests, we first compute the potential at point $\mathbf{x} \in \mathbb{R}^2$ generated by a unit point source placed at point $\mathbf{y}$. In other words, the load $f(\mathbf{x})$ is the *Dirac Delta* function $\delta_{\mathbf{y}} = \delta(\mathbf{x} - \mathbf{y})$ and we want to solve

$$\Delta u(\mathbf{x}) = \delta_{\mathbf{y}}. \tag{2.5}$$

A solution of Eq. (2.5) is called the *fundamental solution* of the laplacian, and it can be determined as follows.

We write Eq. (2.5) in polar coordinates with origin at $\mathbf{y}$. Since this solution is axisymmetric with respect to the source, it is independent of the polar angle θ, thus the laplacian is

$$\Delta u = u_{rr} + \frac{1}{r}\, u_r$$

where $r = |\mathbf{x} - \mathbf{y}|$ is the Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$. The right-hand side vanishes at all points of the plane, except at the origin $r = 0$ (or $\mathbf{x} = \mathbf{y}$), where it has infinite value. Then Eq. (2.5) is written as

$$u_{rr} + \frac{1}{r}\, u_r = 0 \qquad \forall\ r > 0.$$

The change of variables $u_r = v$ gives us

$$v' + \frac{1}{r}\, v = 0 \quad \rightsquigarrow \quad \int \frac{v'}{v} = \int -\frac{1}{r} \quad \rightsquigarrow \quad \log(v) = -\log(r) + c \quad \rightsquigarrow \quad v = u_r = \frac{c}{r}$$

Then

$$u = \int \frac{c}{r} = c_1 \log(r) + c_2.$$

Since we want a particular solution we may set $c_2 = 0$, and to determine c_1 we perform an integration by parts on Eq. (2.5) over a ficticious unit circular domain Ω:

$$\int_\Omega \Delta u\ \varphi\ \mathrm{d}\Omega = \int_{\partial\Omega} \frac{\partial u}{\partial n}\ \varphi\ \mathrm{d}\sigma - \int_\Omega \nabla u \cdot \nabla\varphi\ \mathrm{d}\Omega$$

which, for $\varphi \equiv 1$, reads

$$\int_\Omega \delta(\mathbf{x} - \mathbf{y})\ \mathrm{d}\Omega = \int_{\partial\Omega} \frac{\partial u}{\partial n}\ \mathrm{dS}.$$

Due to the axisymmetric nature of the problem

$$\frac{\partial u}{\partial n} = \frac{\partial u}{\partial r} = \frac{c_1}{r}$$

thus, after using the definition of Delta function and performing an integration in radial coordinates, we are left with

$$1 = 2\pi \int_0^1 \rho\ v(\rho)\ \mathrm{d}\rho = 2\pi \int_0^1 \rho\ \frac{c_1}{\rho}\ \mathrm{d}\rho = 2\pi c_1 \quad \rightsquigarrow \quad c_1 = \frac{1}{2\pi}.$$

Hence, the fundamental solution becomes

$$u = \frac{1}{2\pi}\ \log r,$$

which is also known in the literature as the *free space Green's function*:

$$G(\mathbf{x}, \mathbf{y}) = \frac{1}{2\pi}\ \log|\mathbf{x} - \mathbf{y}| \qquad \forall\ \mathbf{x} \in \mathbb{R}^2. \tag{2.6}$$

In a more advanced mathematical frameowork, one can state the following theorem:

Theorem 2.1. *The function $G(\cdot, \mathbf{y})$ defined in Eq. (2.6) solves*

$$\Delta G(\cdot, \mathbf{y}) = \delta_{\mathbf{y}} \qquad \text{in } \mathcal{D}'(\mathbb{R}^2) \tag{2.7}$$

i.e. in the sense of distributions.

Proof. It suffices to prove the statement in the case $\mathbf{y} \equiv 0$. Let T_G be the linear and continuous functional induced by G:

$$\begin{aligned} T_{_G} : \mathcal{D}(\mathbb{R}^2) &\to \mathbb{R} \\ \varphi &\mapsto \langle T_{_G}, \varphi \rangle = \int_{\mathbb{R}^2} G(\mathbf{x})\ \varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} \end{aligned}$$

The laplacian in distributional sense is such that

$$\langle \Delta T_{_G}, \varphi \rangle = \langle T_{_G}, \Delta\varphi \rangle = \int_{\mathbb{R}^2} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x}.$$

Instead, the Dirac Delta distribution centered at the origin is

$$\begin{aligned} \delta_0 : \mathcal{D}(\mathbb{R}^2) &\to \mathbb{R} \\ \varphi &\mapsto \langle \delta_0, \varphi \rangle = \varphi(0) \end{aligned}$$

Hence, Eq. (2.7) becomes

$$\int_{\mathbb{R}^2} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} = \varphi(0). \tag{2.8}$$

Since G blows up at 0, the idea is to isolate this singularity inside a small ball. Indeed, φ has compact support meaning that there exists an open ball B_R of radius R centered at the origin such that $\operatorname{supp}\varphi \subseteq B_R$. So we fix $\mathcal{E} > 0$ and we decompose the left-hand side as follows:

$$\begin{aligned} \int_{\mathbb{R}^2} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} &= \int_{B_R} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} \\ &= \underbrace{\int_{B_\mathcal{E}} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x}}_{(1)} + \underbrace{\int_{B_R \setminus B_\mathcal{E}} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x}}_{(2)} \end{aligned}$$

We analyze the two integrals separately:

(1) an integration in radial coordinates gives

$$\begin{aligned} \left| \int_{B_\mathcal{E}} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} \right| &= \left| \int_{B_\mathcal{E}} \frac{\log|\mathbf{x}|}{2\pi}\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} \right| \\ &\leq \frac{1}{2\pi}\ \max_{\operatorname{supp}\varphi} |\Delta\varphi| \int_{|\mathbf{x}|<\mathcal{E}} \log|\mathbf{x}|\ \mathrm{d}\mathbf{x} \\ &= \frac{1}{\cancel{2\pi}} \max_{\operatorname{supp}\varphi} |\Delta\varphi|\ \cancel{2\pi} \int_0^\mathcal{E} \rho\ \log\rho\ \mathrm{d}\rho \\ &= \max_{\operatorname{supp}\varphi} |\Delta\varphi|\ \frac{\mathcal{E}^2}{2}\left(\log\mathcal{E} - \frac{1}{2}\right) \xrightarrow{\mathcal{E}\to 0^+} 0 \end{aligned}$$

(2) we apply Green's second identity:

$$\int_{B_R\setminus B_\varepsilon} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} = \int_{B_R\setminus B_\varepsilon} \varphi(\mathbf{x})\ \cancel{\Delta\left(\frac{\log|\mathbf{x}|}{2\pi}\right)}\ \mathrm{d}\Omega(\mathbf{x}) + \\ + \int_{\partial B_\varepsilon \cup \cancel{\partial B_R}} \left[\frac{\log|\mathbf{x}|}{2\pi}\ \frac{\partial\varphi(\mathbf{x})}{\partial n} - \frac{\varphi(\mathbf{x})}{2\pi}\ \frac{\partial \log|\mathbf{x}|}{\partial n}\right] \mathrm{dS}(\mathbf{x}) \\ = \int_{\partial B_\varepsilon} \left[\frac{\log|\mathbf{x}|}{2\pi}\ \nabla\varphi(\mathbf{x})\cdot\mathbf{n} - \frac{\varphi(\mathbf{x})}{2\pi}\ \nabla\log|\mathbf{x}|\cdot\mathbf{n}\right] \mathrm{dS}(\mathbf{x}) =$$

since G is harmonic away from the origin and φ vanishes at radius R. Now $\mathbf{n} = -\frac{\mathbf{x}}{|\mathbf{x}|}$, consequently we have

$$= \underbrace{-\int_{\partial B_\varepsilon} \frac{\log|\mathbf{x}|}{2\pi}\ \nabla\varphi(\mathbf{x})\cdot\frac{\mathbf{x}}{|\mathbf{x}|}\ \mathrm{dS}(\mathbf{x})}_{(3)} + \underbrace{\int_{\partial B_\varepsilon} \frac{\varphi(\mathbf{x})}{2\pi}\ \nabla\log|\mathbf{x}|\cdot\frac{\mathbf{x}}{|\mathbf{x}|}\ \mathrm{dS}(\mathbf{x})}_{(4)}$$

(3) as we already did in (1)

$$\left|-\int_{\partial B_\varepsilon} \frac{\log|\mathbf{x}|}{2\pi}\ \nabla\varphi(\mathbf{x})\cdot\frac{\mathbf{x}}{|\mathbf{x}|}\ \mathrm{dS}(\mathbf{x})\right| \le \max_{\operatorname{supp}\varphi}|\nabla\varphi|\ \frac{\log\mathcal{E}}{\cancel{2\pi}} \underbrace{\int_{\partial B_\varepsilon} \mathrm{dS}(\mathbf{x})}_{\cancel{2\pi}\,\mathcal{E}} \xrightarrow{\varepsilon\to 0^+} 0$$

(4) here we simply compute the derivative

$$\int_{\partial B_\varepsilon} \frac{\varphi(\mathbf{x})}{2\pi}\ \nabla\log|\mathbf{x}|\cdot\frac{\mathbf{x}}{|\mathbf{x}|}\ \mathrm{dS}(\mathbf{x}) = \int_{\partial B_\varepsilon} \frac{\varphi(\mathbf{x})}{2\pi}\frac{1}{\mathbf{x}}\cdot\frac{\mathbf{x}}{|\mathbf{x}|}\ \mathrm{dS}(\mathbf{x}) \\ = \frac{1}{2\pi\mathcal{E}}\int_{\partial B_\varepsilon} \varphi(\mathbf{x})\ \mathrm{dS}(\mathbf{x})$$

that is the integral mean value of φ. Then $\exists\,\xi\in\overline{\partial B_\varepsilon}$ such that

$$\varphi(\xi) = \frac{1}{|\partial B_\varepsilon|}\int_{\partial B_\varepsilon} \varphi\ \mathrm{dS} \xrightarrow{\varepsilon\to 0^+} \varphi(0)$$

because φ is continuous over ∂B_ε.

To wrap up, we have shown that

$$\left|\int_{\mathbb{R}^2} G(\mathbf{x})\ \Delta\varphi(\mathbf{x})\ \mathrm{d}\mathbf{x} - \varphi(0)\right| \xrightarrow{\varepsilon\to 0^+} 0$$

therefore we proved Eq. (2.8), as asserted. □

2.2.2. BIE for the Laplace Equation

Here we derive the solution of the Laplace equation with mixed boundary conditions:

$$\begin{cases} \Delta u = 0 & \text{in } \Omega, \\ u = \overline{u} & \text{on } \Gamma_1, \\ \dfrac{\partial u}{\partial n} = \overline{u}_n & \text{on } \Gamma_2. \end{cases} \tag{2.9}$$

We can multiply the Laplace equation by an arbitrary test function φ and integrate by parts twice:

$$\begin{aligned} 0 = \int_\Omega \Delta u \; \varphi \; \mathrm{d}\Omega &= \int_\Gamma \frac{\partial u}{\partial n} \; \varphi \; \mathrm{dS} - \int_\Omega \nabla u \; \nabla \varphi \; \mathrm{d}\Omega \\ &= \int_\Gamma \frac{\partial u}{\partial n} \; \varphi \; \mathrm{dS} - \int_\Gamma \frac{\partial \varphi}{\partial n} \; u \; \mathrm{dS} + \int_\Omega \Delta \varphi \; u \; \mathrm{d}\Omega \end{aligned}$$

(one could directly apply the Green's second identity). Choosing $\varphi \equiv G$ the free space Green's function defined in Eq. (2.6) yields

$$0 = \int_\Omega \Delta G(\mathbf{x}, \mathbf{y}) \; u(\mathbf{x}) \; \mathrm{d}\Omega(\mathbf{x}) + \int_\Gamma \left[\frac{\partial u}{\partial n}(\mathbf{x}) \; G(\mathbf{x}, \mathbf{y}) - \frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y}) \; u(\mathbf{x}) \right] \mathrm{dS}(\mathbf{x}).$$

Since we know that $\Delta G(\mathbf{x}, \mathbf{y}) = \delta_{\mathbf{y}}$,

$$0 = \int_\Omega \delta(\mathbf{x} - \mathbf{y}) \; u(\mathbf{x}) \; \mathrm{d}\Omega(\mathbf{x}) + \int_\Gamma \left[\frac{\partial u}{\partial n}(\mathbf{x}) \; G(\mathbf{x}, \mathbf{y}) - \frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y}) \; u(\mathbf{x}) \right] \mathrm{dS}(\mathbf{x})$$

hence, by the definition of Dirac Delta, we are left with

$$u(\mathbf{y}) = - \int_\Gamma \left[\frac{\partial u}{\partial n}(\mathbf{x}) \; G(\mathbf{x}, \mathbf{y}) - \frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y}) \; u(\mathbf{x}) \right] \mathrm{dS}(\mathbf{x}) \qquad \forall \; \mathbf{y} \in \Omega. \tag{2.10}$$

This expression is the *integral representation* of the solution for the Laplace equation at any point $\mathbf{y}$ inside the domain Ω in terms of the boundary values of u and its normal derivative $\partial u / \partial n$.

It is apparent from the mixed boundary conditions that only one of the quantities u or $\partial u / \partial n$ is prescribed at a point $\mathbf{x}$ on the boundary. Consequently, it is not yet possible to determine the solution from the integral representation (2.10). Therefore, we let $\mathbf{y}$ lie on the boundary Γ: we notiche that the kernels $G(\mathbf{x}, \mathbf{y})$ and $\frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y})$ become weakly singular (but integrable) and singular respectively. Hence, considering the Cauchy Principal Value (CPV) of the $\frac{\partial G}{\partial n}$ singular integral, we can write

$$\alpha(\mathbf{y}) u(\mathbf{y}) = - \int_\Gamma \frac{\partial u}{\partial n}(\mathbf{x}) \; G(\mathbf{x}, \mathbf{y}) \; \mathrm{d}S(\mathbf{x}) + \int_\Gamma^{\mathrm{PV}} \frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y}) \; u(\mathbf{x}) \; \mathrm{d}S(\mathbf{x}), \tag{2.11}$$

where the coefficient $\alpha(\mathbf{y})$ is obtained fromt the CPV evaluation of the singular integral, and it represents the fraction of solid angle (see Figure 2.5) with which the domain Ω is seen from the boundary point $\mathbf{x}$:

$$\begin{cases} 1 & \text{for } \mathbf{y} \text{ inside } \Omega, \\ \dfrac{\theta_1 - \theta_2}{2\pi} & \text{for } \mathbf{y} \text{ on the boundary } \Gamma, \\ 0 & \text{for } \mathbf{y} \text{ outside } \Omega. \end{cases}$$

In particular, $\alpha = 1/2$ for smooth boundary points:

$$\frac{1}{2}\, u(\mathbf{y}) = -\int_\Gamma \frac{\partial u}{\partial n}(\mathbf{x})\; G(\mathbf{x},\mathbf{y})\; \mathrm{d}S(\mathbf{x}) + \int_\Gamma \frac{\partial G}{\partial n}(\mathbf{x},\mathbf{y})\; u(\mathbf{x})\; \mathrm{d}S(\mathbf{x}). \tag{2.12}$$

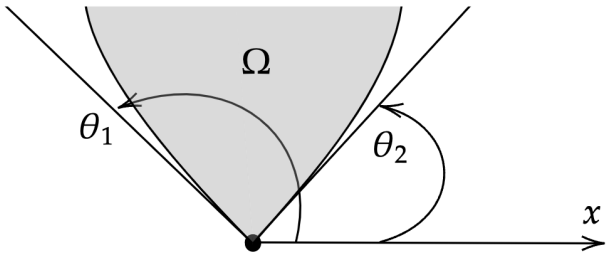


Figure 2.5: Solid angle related to a corner point of a nonsmooth boundary.

Equation (2.12) is an integral equation on the boundary Γ, that is a *boundary integral equation* with which it is possible to derive the potential where its normal derivative is known, and viceversa.

Indeed, assuming smooth boundary, by explicitly writing the boundary conditions we obtain two separate equations

$$\frac{1}{2}\,\overline{u} = -\int_\Gamma \left(G\,\frac{\partial u}{\partial n} - \overline{u}\,\frac{\partial G}{\partial n} \right) \mathrm{d}S, \tag{2.13}$$

$$\frac{1}{2}\,u = -\int_\Gamma \left(G\,\overline{u}_n - u\,\frac{\partial G}{\partial n} \right) \mathrm{d}S. \tag{2.14}$$

2.2.3. BIE for the Poisson Equation

We briefly mention that in the case of Poisson equation, the BIE is turned into

$$\frac{1}{2}\, u = u_0 + u_1$$

where u_0 is the already found solution of the homogeneous equation (Laplace equation) and u_1 is a particular solution, i.e. any function that satisfies only the governing equation $\Delta u_1 = f$ independently of boundary conditions. Hence, by defintion of the fundamental solution, we simply have

$$u_1 = \int_\Omega G\, f \, \mathrm{d}\Omega. \tag{2.15}$$

2.3. Numerical Implementation of the BEM

Let us consider the case where $f = 0$. The quintessence of the BEM is to discretize the boundary Γ into a finite number of segments which are called *boundary elements*. Two approximations are made over each of these elements. One concerns the geometry of the boundary, while the other has to do with the variation of the unknown boundary quantity over the element. Here we choose the simplest approach, which means that our boundary element is a *constant element*: the boundary segment is approximated by a straight line which connects the two *end points*, and then we place a *nodal point* at the midpoint of the line. In other words, the boundary quantity is assumed to be constant along the element and equal to its value at the nodal point (see Figure 2.6).

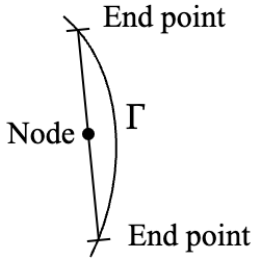


Figure 2.6: Approximation of the boundary segment by a constant element.

2.3.1. BEM with constant elements for the Laplace Equation

The boundary Γ is discretized into N constant elements numbered in counter-clockwise sense. Therefore the discretized version of Eq. (2.12) for a given point $\mathbf{y}_i$ on Γ is

$$\frac{1}{2}\, u^i = -\sum_{j=1}^{N} \int_{\Gamma_j} \frac{\partial u}{\partial n}(\mathbf{x})\; G(\mathbf{x}, \mathbf{y}_i)\; \mathrm{d}S(\mathbf{x}) + \sum_{j=1}^{N} \int_{\Gamma_j} \frac{\partial G}{\partial n}(\mathbf{x}, \mathbf{y}_i)\; u(\mathbf{x})\; \mathrm{d}S(\mathbf{x}). \tag{2.16}$$

Since the values of u and $\partial u/\partial n$ are constant on each element, they can be moved outside the integral. Then Eq. (2.16) may be written as

$$-\frac{1}{2}\, u^i + \sum_{j=1}^{N} \underbrace{\left(\int_{\Gamma_j} \frac{\partial G}{\partial n}\, \mathrm{d}S \right)}_{\hat{H}_{ij}} u^j = \sum_{j=1}^{N} \underbrace{\left(\int_{\Gamma_j} G\, \mathrm{d}S \right)}_{G_{ij}} u_n^j \tag{2.17}$$

Simply setting

$$H_{ij} = \hat{H}_{ij} - \frac{1}{2}\,\delta_{ij},$$

where δ_{ij} is the *Kronecker delta*, Eq. (2.17) becomes

$$\sum_{j=1}^{N} H_{ij}\, u^j = \sum_{j=1}^{N} G_{ij}\, u_n^j. \tag{2.18}$$

We apply Equation (2.18) to all nodes $\mathbf{y}_i$ for $i = 1, \ldots, N$ and we finally obtain a system of N linear algebraic equations, which are arranged in matrix form

$$Hu = Gu_n, \qquad H,\, G \in \mathbb{R}^{N\times N}, \qquad u,\, u_n \in \mathbb{R}^{N\times 1}. \tag{2.19}$$

Since we have mixed boundary conditions (2.9), we need to decouple the system by separating the unknown from the known quantities. This leads to a final $N \times N$ system in the form

$$Ax = b, \tag{2.20}$$

where $x = (u \text{ on } \Gamma_2,\ u_n \text{ on } \Gamma_1)'$ groups the unknowns.

Once the boundary quantities u and u_n are known, the solution u can be computed at any point $\mathbf{y}$ inside the domain Ω using Eq. (2.11)

$$u(\mathbf{y}) = \sum_{j=1}^{N} \hat{H}_{ij}\, u^j - \sum_{j=1}^{N} G_{ij}\, u_n^j \tag{2.21}$$

and by direct differentiation we can obtain its gradient.

2.3.2. Line integrals and BEM for the Poisson Equation

We mention that the line integrals G_{ij} and $\hat{H}_{ij}$ are evaluated numerically using a standard Gaussian quadrature. Moreover, the Poisson version of system (2.19) is

$$Hu + F = Gu_n, \tag{2.22}$$

where the term F contains the evaluation of the domain integrals (2.15) by employing two-dimensional Gaussian integration. For further explanations, see [1].

On the basis of the analysis presented so far, we are going to present and solve three problems ruled by the Laplace equation. Then we will compare our MATLAB and Abaqus results, as well as with the ones obtained using FORTRAN in [1].

2.4. Problem 1

The scope of this problem is to illustrate the MATLAB implementation of the BEM by solving a simple potential problem for the Laplace equation in the unit square domain Ω under mixed boundary conditions as shown in Figure 2.8.

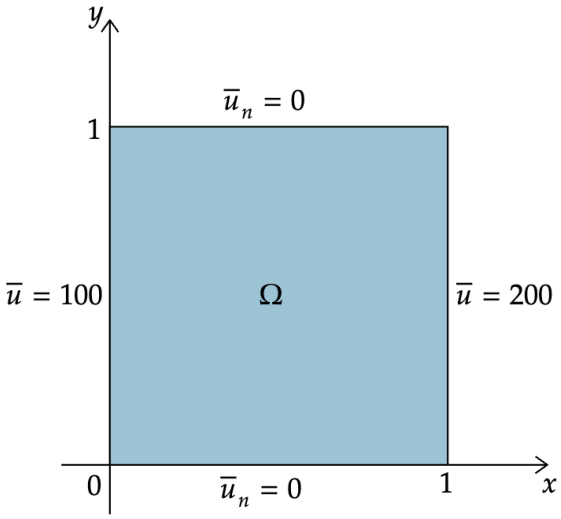


Figure 2.7: Square domain Ω and boundary conditions of Problem 1.

2.4.1. MATLAB Implementation

We start with a very coarse grid, namely made with a number of elements N = 16. In Table 2.1 we explain the other inputs:

Variable	Meaning
N	Number of boundary elements and boundary nodes
IN	Number of internal points where the solution is computed
XL	One-dimensional array containing the x coordinates of the extreme points of all the elements
YL	One-dimensional array containing the y coordinates of the extreme points of all the elements
XIN	One-dimensional array containing the x coordinates of the internal points at which te values of u are computed
YIN	One-dimensional array containing the y coordinates of the internal points at which te values of u are computed
INDEX	One-dimensional boolean array in which a type of boundary condition is assigned to the nodes
UB	One-dimensional array containing the boundary values of u, if INDEX=0, or $\partial u/\partial n$, if INDEX=1

Table 2.1: Inputs for the MATLAB program.

We immediately compute the x and y coordinates of all the boundary nodes thanks to the `MIDPOINTS` function (Algorithm 2.1), and we store them into XM and YM arrays.

Hence, the initial discretization, as the `INPUT` function (Algorithm that just prints the input data) can confirm, looks like this:

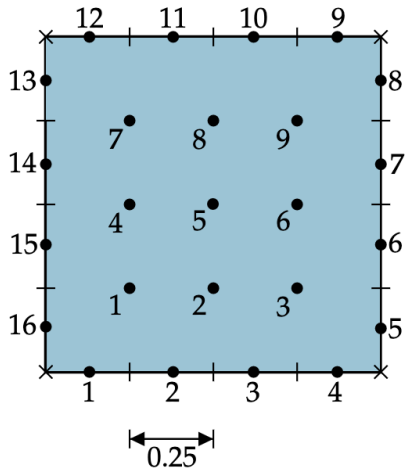


Figure 2.8: Boundary element discretization and internal points of Problem 1.

Then we compute the G matrix with Algorithm 2.2 and the H matrix with Algorithm 2.3. The line intelgrals are computed locally by using the auxiliary Algorithms 2.4, 2.5, 2.6.

The function `ABMATR` (Algorithm 2.7) generates the matrix A and the vector b of Eq. (2.20). Thus, we solve the system with the *backslash* command.

With Algorithm 2.8 we rearrange the solutions according to the INDEX vector, and we finally obtain the solutions vector UB and UNB containing the values of u and $\partial u/\partial n$ on the boundary points.

Finally, we compute the internal values of u thanks to Algorithm 2.9 and we print the results with the `OUTPUT` function.

To sum up, the pseudocode of our program is the following:

$$\begin{aligned} &\text{INPUT} \rightarrow \text{MIDPOINTS} \rightarrow \text{GMATR} \rightarrow \text{HMATR} \rightarrow \text{ABMATR} \\ &\rightarrow \textit{solve the system (backslash)} \rightarrow \text{REORDER} \rightarrow \text{UINTER} \rightarrow \text{OUTPUT} \end{aligned} \tag{2.23}$$

The script we run is `main21` (Algorithm 2.10), and we obtain the following output:

```
**********************************************************************
RESULTS
**********************************************************************

BOUNDARY NODES:

NODE        XM           YM           U            U_n
 1        0.12500      0.00000     111.87694      0.00000
 2        0.37500      0.00000     137.32269      0.00000
 3        0.62500      0.00000     162.67731      0.00000
 4        0.87500      0.00000     188.12306      0.00000
 5        1.00000      0.12500     200.00000    105.52311
 6        1.00000      0.37500     200.00000     98.41687
 7        1.00000      0.62500     200.00000     98.41687
 8        1.00000      0.87500     200.00000    105.52311
 9        0.87500      1.00000     188.12306      0.00000
 10       0.62500      1.00000     162.67731      0.00000
 11       0.37500      1.00000     137.32269      0.00000
 12       0.12500      1.00000     111.87694      0.00000
 13       0.00000      0.87500     100.00000   -105.52311
 14       0.00000      0.62500     100.00000    -98.41687
 15       0.00000      0.37500     100.00000    -98.41687
 16       0.00000      0.12500     100.00000   -105.52311

INTERNAL POINTS:

POINT       XIN          YIN          U
  1       0.25000      0.25000    124.88609
  2       0.50000      0.25000    150.00000
  3       0.75000      0.25000    175.11391
  4       0.25000      0.50000    124.95125
  5       0.50000      0.50000    150.00000
  6       0.75000      0.50000    175.04875
  7       0.25000      0.75000    124.88609
  8       0.50000      0.75000    150.00000
  9       0.75000      0.75000    175.11391
```

These results perfectly match the ones obtained with FORTRAN in [1]. Both of them can be compared with the exact solution $u(x, y) = 100(1 + x)$, showing a rapid convergence.

2.4.2. MATLAB scripts

Algorithm 2.1: MIDPOINTS

```
function [XM, YM] = MIDPOINTS(XL, YL, N)
xl = [XL; XL(1)];
yl = [YL; YL(1)];
XM = zeros(N,1);
YM = zeros(N,1);
for i = 1:N
    XM(i) = (xl(i)+xl(i+1))/2;
    YM(i) = (yl(i)+yl(i+1))/2;
end
end
```

Algorithm 2.2: GMATR

```
function G = GMATR(XL, YL, XM, YM, N)
G = zeros(N,N);
for i = 1:N
    for j = 1:N
        if i ~= j % off-diagonal elements of G
            if j==N
                G(i,j) = RLINTC(XM(i),YM(i),XL(j),YL(j),XL(1),YL(1));
            else
                G(i,j) = RLINTC(XM(i),YM(i),XL(j),YL(j),XL(j+1),YL(j+1));
            end
        elseif i == j % diagonal elements of G
            if j==N
                G(i,j) = SLINTC(XL(j),YL(j),XL(1),YL(1));
            else
                G(i,j) = SLINTC(XL(j),YL(j),XL(j+1),YL(j+1));
            end
        end
    end
end
end
```

Algorithm 2.3: HMATR

```
function H = HMATR(XL, YL, XM, YM, N)
H = zeros(N,N);
for i = 1:N
    for j = 1:N
        if i ~= j % off-diagonal elements of H
            if j==N
                H(i,j) = DALPHA(XM(i),YM(i),XL(j),YL(j),XL(1),YL(1));
```

```
            else
                H(i,j) = DALPHA(XM(i),YM(i),XL(j),YL(j),XL(j+1),YL(j+1));
            end
        elseif i == j % diagonal elements of H
            H(i,j) = -0.5;
        end
    end
end
end
```

Algorithm 2.4: RLINTC

```
function result = RLINTC(x0, y0, x1, y1, x2, y2)
% Define Gauss integration points and weights
xi = [-0.86113631,-0.33998104,0.33998104,0.86113631];
wg = [0.34785485,0.65214515,0.65214515,0.34785485];
xc = zeros(4,1);
yc = zeros(4,1);
% Compute constants
ax = (x2-x1)/2;
ay = (y2-y1)/2;
bx = (x2+x1)/2;
by = (y2+y1)/2;
% Compute the line integral
result = 0;
for i = 1:4
    xc(i) = ax*xi(i) + bx;
    yc(i) = ay*xi(i) + by;
    ra = sqrt((xc(i)-x0)^2 + (yc(i)-y0)^2);
    result = result + log(ra)*wg(i);
end
sl = 2*sqrt(ax^2 + ay^2);
result = result*sl/(4*pi);
end
```

Algorithm 2.5: SLINTC

```
function result = SLINTC(x1, y1, x2, y2)
% Compute constants
ax = (x2-x1)/2;
ay = (y2-y1)/2;
sl = sqrt(ax^2 + ay^2);
% Compute the line integral
result = sl*(log(sl)-1)/pi;
end
```

Algorithm 2.6: DALPHA

```
function result = DALPHA(x0, y0, x1, y1, x2, y2)
% Compute constants
dy1 = y1 - y0;
dx1 = x1 - x0;
dy2 = y2 - y0;
dx2 = x2 - x0;
dl1 = sqrt(dx1^2 + dy1^2);
cos1 = dx1/dl1;
sin1 = dy1/dl1;
dx2r = dx2*cos1 + dy2*sin1;
dy2r = -dx2*sin1 + dy2*cos1;
da = atan2(dy2r,dx2r);
% Compute the line integral
result = da/(2*pi);
end
```

Algorithm 2.7: ABMATR

```
function [A, UNB] = ABMATR(G, H, UB, INDEX)
n = size(G,1);
A = zeros(n,n);
UNB = zeros(n,1);
% Reorder the columns of the system of equations and store them in A
for j = 1:n
    if INDEX(j) == 0
        A(:,j) = -G(:,j);
    else
        A(:,j) = H(:,j);
    end
end
% Compute the right-hand side vector and store it in UNB
for i = 1:n
    for j = 1:n
        if INDEX(j) == 0
            UNB(i) = UNB(i) - H(i,j)*UB(j);
        else
            UNB(i) = UNB(i) + G(i,j)*UB(j);
        end
    end
end
end
```

Algorithm 2.8: REORDER

```
function [UB, UNB] = REORDER(UB, UNB, INDEX)
```

```
n = length(UB);
% Rearrange the arrays
for i = 1:n
    if INDEX(i) ~= 0
        ch = UB(i);
        UB(i) = UNB(i);
        UNB(i) = ch;
    end
end
end
```

Algorithm 2.9: UINTER

```
function UIN = UINTER(XL, YL, XIN, YIN, UB, UNB, N, IN)
UIN = zeros(IN, 1);
for k = 1:IN
    for j = 1:N
        if j==N
            resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(1), YL(1));
            resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j), XL(1), YL(1));
        else
            resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(j+1), YL(j+1))
   ;
            resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j), XL(j+1), YL(j+1))
   ;
        end
        UIN(k) = UIN(k) + resh*UB(j) - resg*UNB(j);
    end
end
end
```

Algorithm 2.10: main21

```
%% Problem 1 - Benchmark
clear; clc;

% Set the maximum dimensions
N = 16;
IN = 9;

% Set data
XL = [0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25; 0; 0; 0; 0];
YL = [0; 0; 0; 0; 0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25];
XIN = [0.25; 0.5; 0.75; 0.25; 0.5; 0.75; 0.25; 0.5; 0.75];
YIN = [0.25; 0.25; 0.25; 0.5; 0.5; 0.5; 0.75; 0.75; 0.75];
INDEX = [1; 1; 1; 1; 0; 0; 0; 0; 1; 1; 1; 1; 0; 0; 0; 0];
```

```
UB = [0; 0; 0; 0; 200; 200; 200; 200; 0; 0; 0; 0; 100; 100; 100; 100];

% Compute midpoints
[XM, YM] = MIDPOINTS(XL, YL, N);

% Print data
INPUT(XL, YL, XM, YM, XIN, YIN, INDEX, UB, N, IN);

% Compute the G matrix
G = GMATR(XL, YL, XM, YM, N);

% Compute the H matrix
H = HMATR(XL, YL, XM, YM, N);

% Form the system of equations AX=B
[A, UNB] = ABMATR(G, H, UB, INDEX);

% Solve the system of equations
UNB = A\UNB;

% Form the vectors U and UN of all the boundary values
[UB, UNB] = REORDER(UB, UNB, INDEX);

% Compute the values UIN of u at the internal points
UIN = UINTER(XL, YL, XIN, YIN, UB, UNB, N, IN);

% Print results
OUTPUT(XM, YM, XIN, YIN, UB, UNB, UIN, N, IN);
```

2.5. Problem 2

The second 2D steady-state heat conduction problem we aim to address involves a square plate made of a thermally isotropic material, with a side length of 1 m and a negligible thickness of 1 mm. The temperature distribution $T = T(x, y)$ is specified along the entire boundary Γ, providing the necessary boundary conditions for the analysis.

We are going to solve this problem using both MATLAB and Abaqus softwares. Specifically, we want to compare the solutions obtained from the FEM and the BEM at the nine internal nodes illustrated in the following figure:

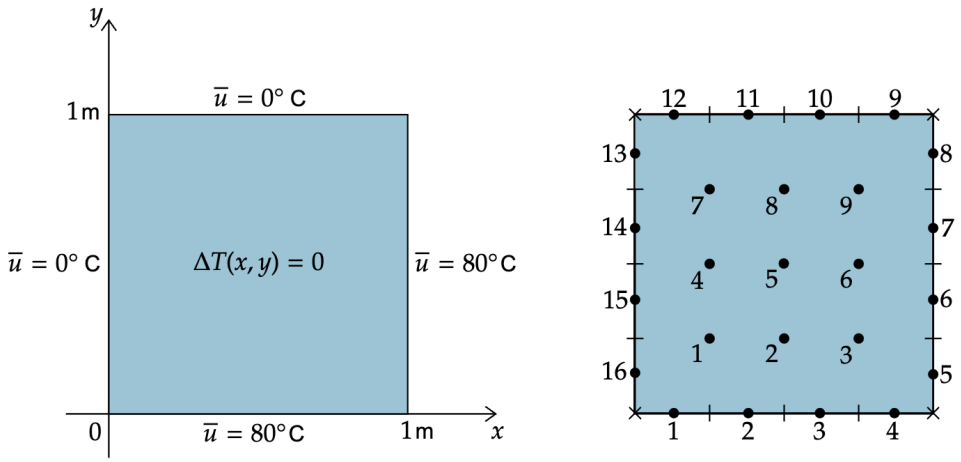


Figure 2.9: Left: square domain Ω and boundary conditions of Problem 2. Right: boundary element discretization and internal points of Problem 2.

2.5.1. MATLAB Implementation

As we can see from the previous picture, the set up of this problem and Problem 1 are equal. Therefore, we simply run the script `main22` (Algorithm 2.11), and we obtain the following output:

```
**********************************************************************
RESULTS
**********************************************************************

BOUNDARY NODES:

NODE         XM           YM           U           U_n
 1        0.12500      0.00000     80.00000    470.36316
 2        0.37500      0.00000     80.00000    116.43137
 3        0.62500      0.00000     80.00000     66.83419
 4        0.87500      0.00000     80.00000     18.38842
 5        1.00000      0.12500     80.00000     18.38842
 6        1.00000      0.37500     80.00000     66.83419
```

```
  7        1.00000        0.62500       80.00000      116.43137
  8        1.00000        0.87500       80.00000      470.36316
  9        0.87500        1.00000        0.00000     -470.36316
  10       0.62500        1.00000        0.00000     -116.43137
  11       0.37500        1.00000        0.00000      -66.83419
  12       0.12500        1.00000        0.00000      -18.38842
  13       0.00000        0.87500        0.00000      -18.38842
  14       0.00000        0.62500        0.00000      -66.83419
  15       0.00000        0.37500        0.00000     -116.43137
  16       0.00000        0.12500        0.00000     -470.36316

 INTERNAL POINTS:

 POINT         XIN            YIN            U
   1         0.25000        0.25000       40.00000
   2         0.50000        0.25000       57.88403
   3         0.75000        0.25000       69.15628
   4         0.25000        0.50000       22.11597
   5         0.50000        0.50000       40.00000
   6         0.75000        0.50000       57.88403
   7         0.25000        0.75000       10.84372
   8         0.50000        0.75000       22.11597
   9         0.75000        0.75000       40.00000
```

2.5.2. MATLAB scripts

Algorithm 2.11: main22

```
%% Problem 2 - 2D steady state heat conduction problem
clear; clc;

% Set the maximum dimensions
N = 16;
IN = 9;

% Set data
XL = [0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25; 0; 0; 0; 0];
YL = [0; 0; 0; 0; 0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25];
XIN = [0.25; 0.5; 0.75; 0.25; 0.5; 0.75; 0.25; 0.5; 0.75];
YIN = [0.25; 0.25; 0.25; 0.5; 0.5; 0.5; 0.75; 0.75; 0.75];
INDEX = zeros(N,1);
UB = zeros(N,1); UB(1:8) = 80;

% Compute midpoints
[XM, YM] = MIDPOINTS(XL, YL, N);
```

```
% Print data
INPUT(XL, YL, XM, YM, XIN, YIN, INDEX, UB, N, IN);

% Compute the G matrix
G = GMATR(XL, YL, XM, YM, N);

% Compute the H matrix
H = HMATR(XL, YL, XM, YM, N);

% Form the system of equations AX=B
[A, UNB] = ABMATR(G, H, UB, INDEX);

% Solve the system of equations
UNB = A\UNB;

% Form the vectors U and UN of all the boundary values
[UB, UNB] = REORDER(UB, UNB, INDEX);

% Compute the values UIN of u at the internal points
UIN = UINTER(XL, YL, XIN, YIN, UB, UNB, N, IN);

% Print results
OUTPUT(XM, YM, XIN, YIN, UB, UNB, UIN, N, IN);
```

2.5.3. Abaqus Implementation

In this section we solve Problem 2 through the Abaqus Software. The chosen units of measure are mm for length, °C for temperature, and W for power (energy per unit time). Consequently, the thermal conductivity of the material is given by:

$$k = 1 \frac{\mathrm{W}}{\mathrm{m}\ ^\circ\mathrm{C}} = 1000 \frac{\mathrm{W}}{\mathrm{mm}\ ^\circ\mathrm{C}} \tag{2.24}$$

Part module

We analyze the plate as a 2D planar deformable shell body of size 2000:

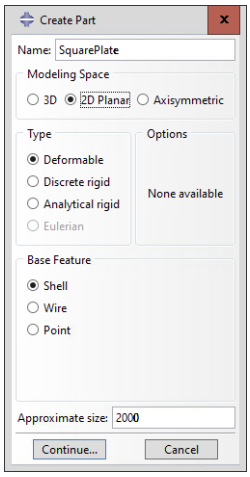


Figure 2.10: Create part.

We obtain:

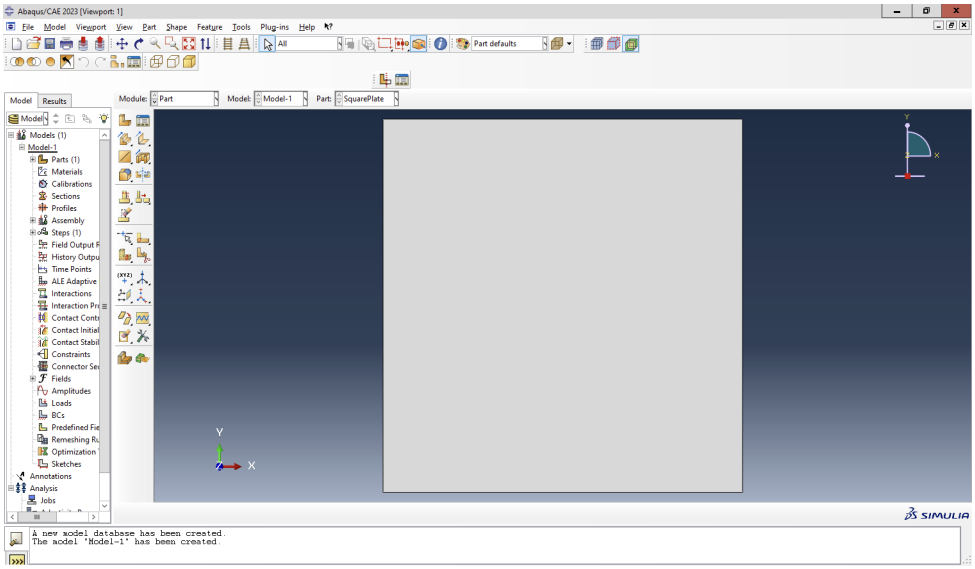


Figure 2.11: Final result of Part module.

Property module

We create a new material with conductivity k:

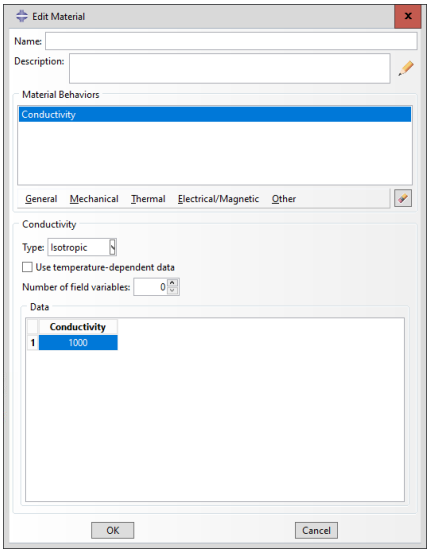


Figure 2.12: Create Material.

Then we define a solid and homogeneous section

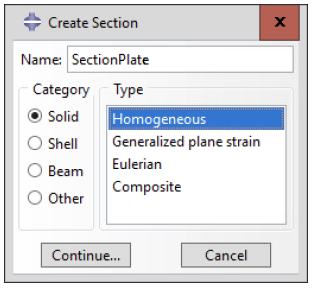


Figure 2.13: Create Section.

and we assign the section by selecting the whole plate, as Figure 2.14 shows.

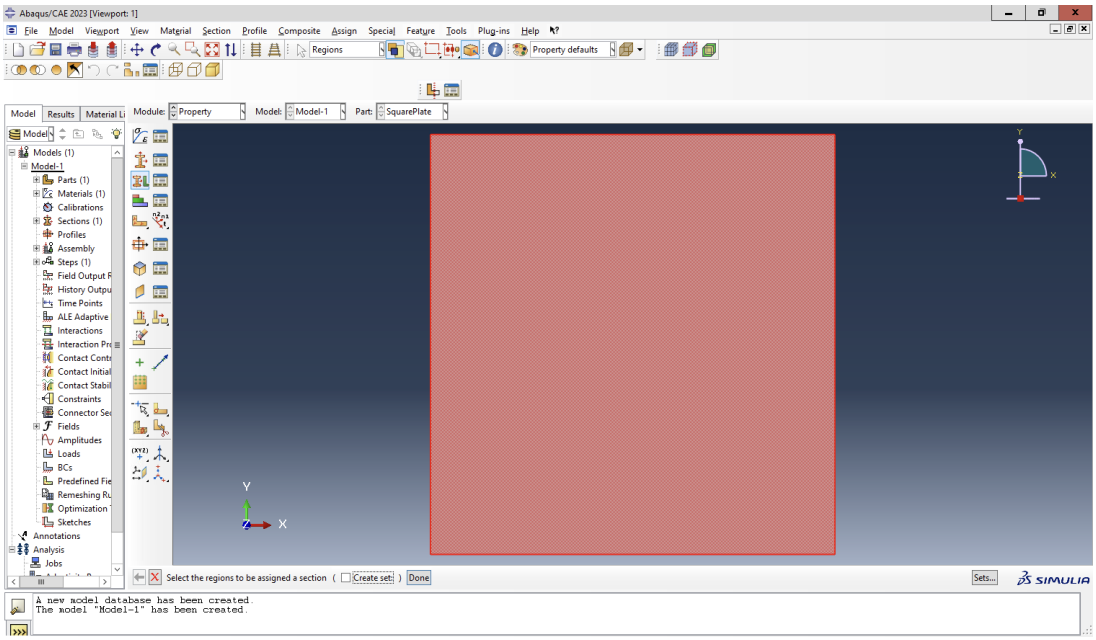


Figure 2.14: Assign Section.

Assembly module

Geometry and material have been defined at this stage. However, we need to create a dependent instance in order to apply boundary conditions:

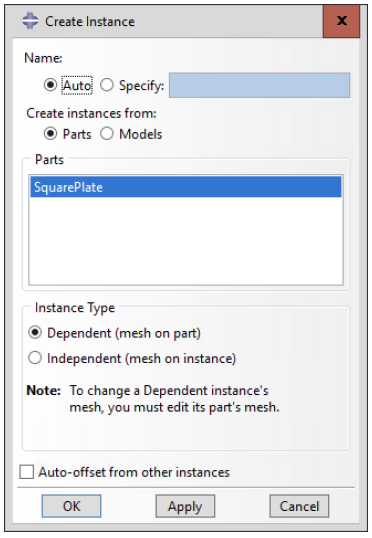


Figure 2.15: Create Instance.

Thus we get:

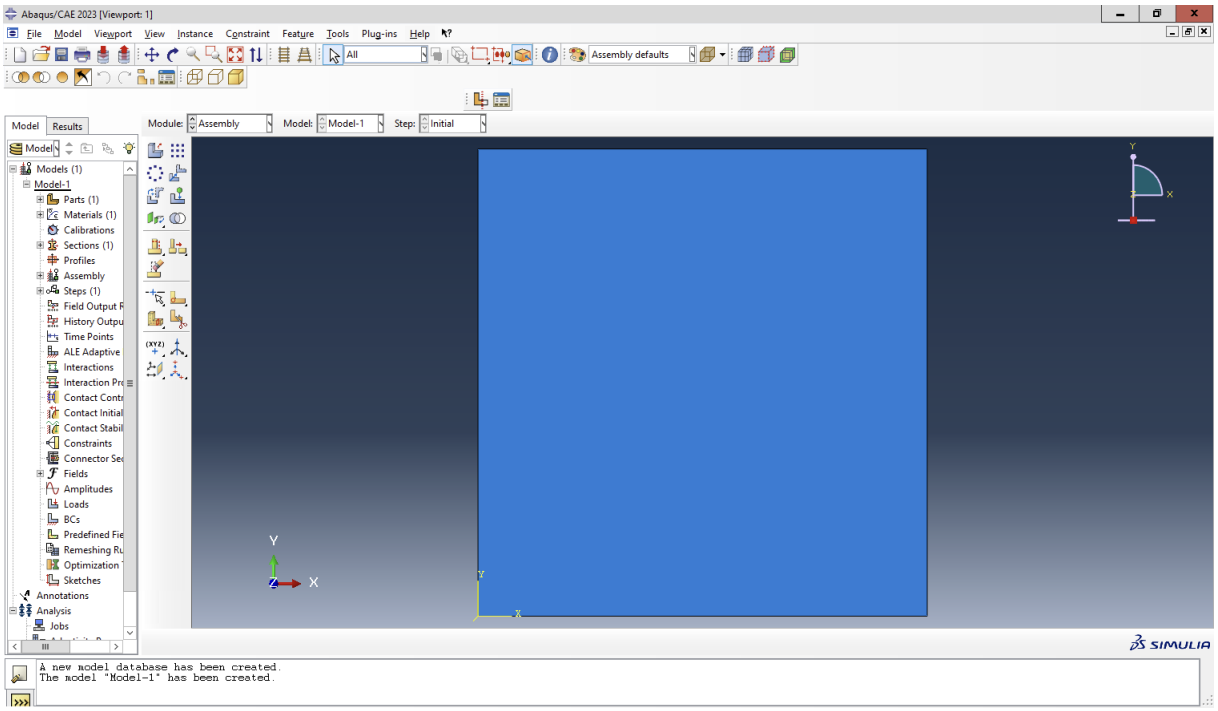


Figure 2.16: Final result of Assembly module.

Step module

In this module we define the type of analysis we need: a steady state heat transfer step.

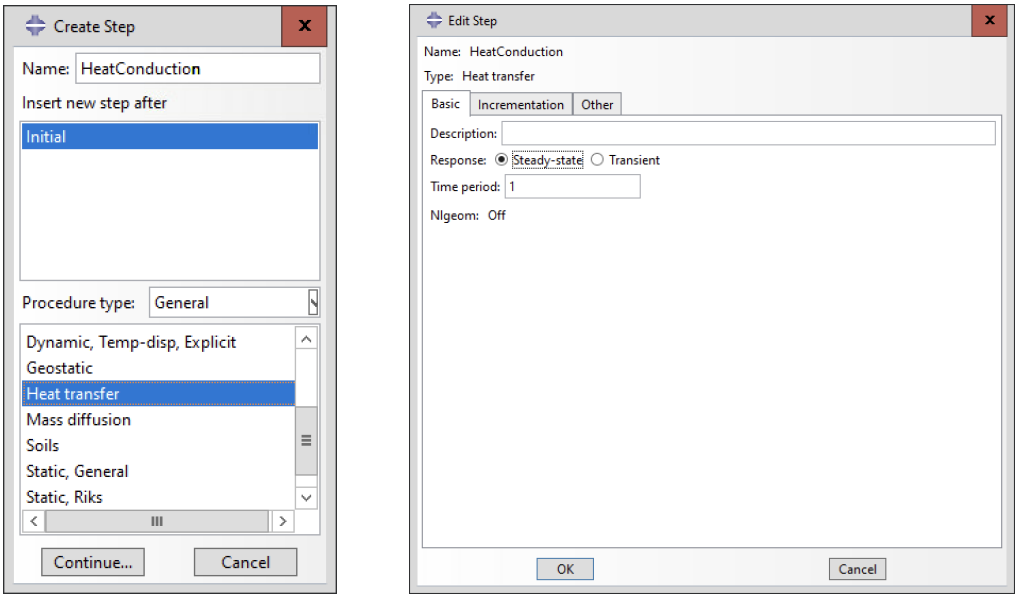


Figure 2.17: Create and edit step: heat transfer steady state analysis.

Load module

We create the two different boundary conditions to be imposed on Γ:

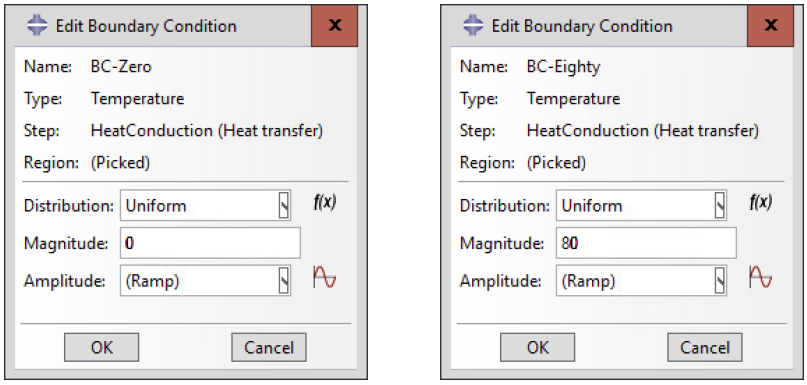


Figure 2.18: Create and edit boundary conditions.

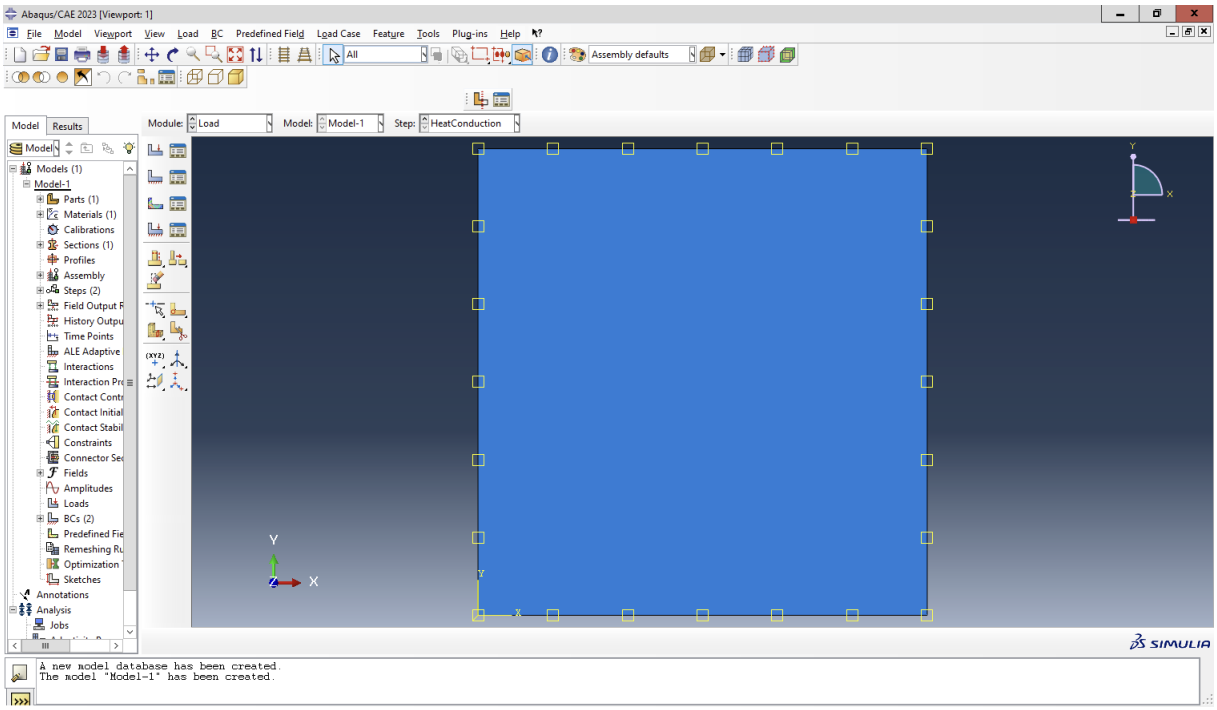


Figure 2.19: Final result of Load module.

Mesh module

Here we generate a mesh of 64 four-node linear heat transfer quadrilateral finite elements:

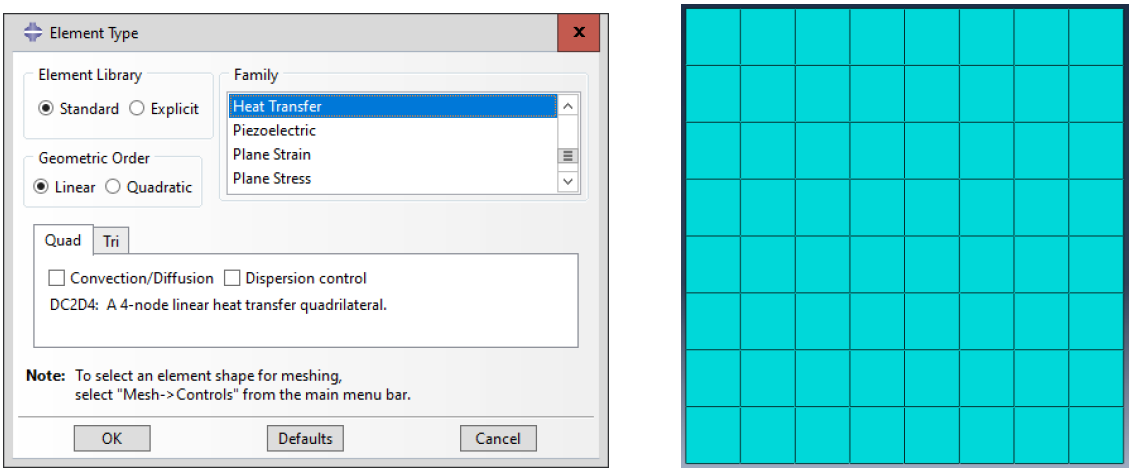


Figure 2.20: Element type and generated mesh.

Job module

Finally, we can submit a standard job:

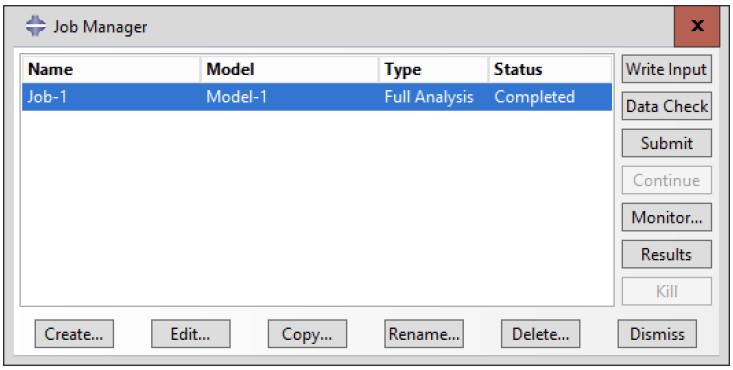


Figure 2.21: Job Manager.

Visualization module

Results of the FEM analysis can be visualized in this module. We frist display the nodal temperature (NT11) distribution:

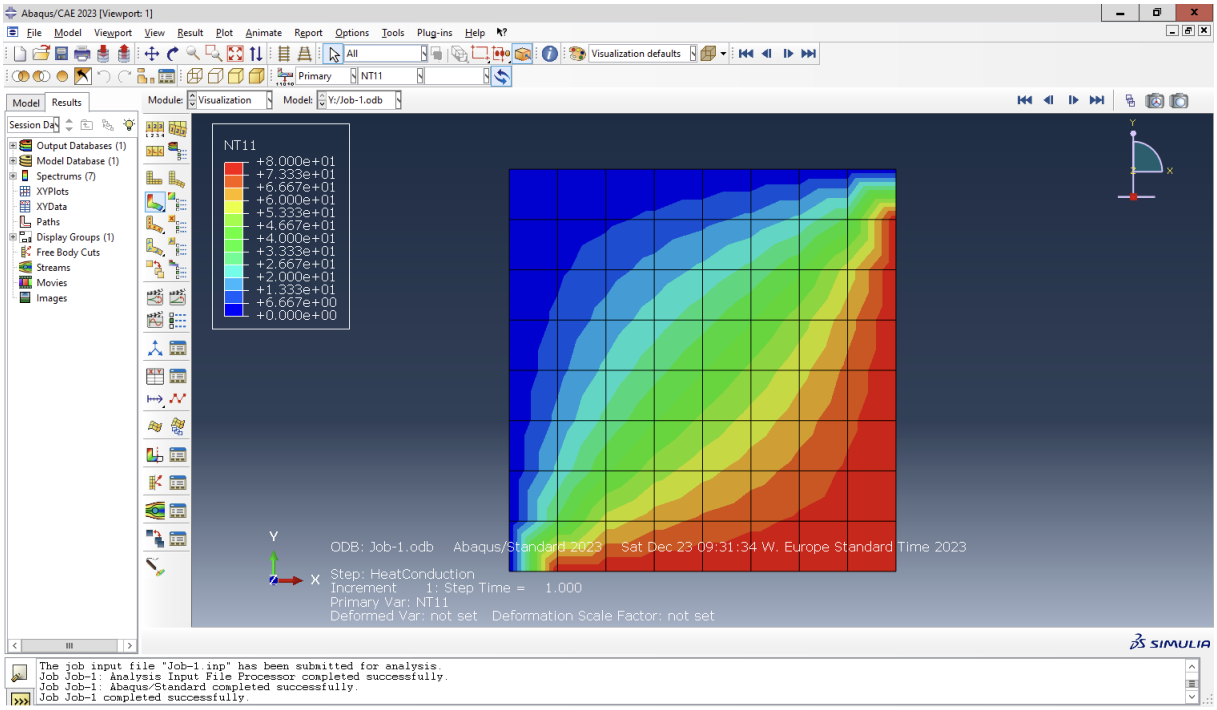


Figure 2.22: Nodal temperature.

Then we proceed by selecting the nodal temperature of the nine internal node we are interested in:

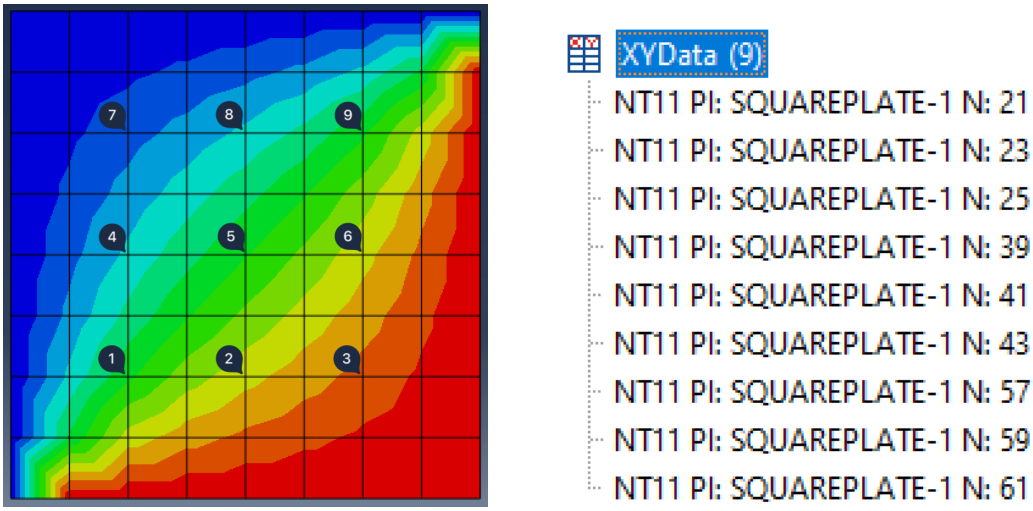


Figure 2.23: Nodal temperature of internal points.

The numerical values are contained in Table 2.2.

2.5.4. Comparison between MATLAB and Abaqus

We now present the comparison of the results obtained from solving Problem 2 using MATLAB and Abaqus software. Both the Finite Element Method and the Boundary Element Method were employed to analyze the steady state heat conduction in a square plate with specified boundary conditions. The comparison focuses on the temperature distribution at nine selected internal nodes, as depicted in Fig. 2.9 and Fig. 2.23:

Internal Point label	Abaqus label	Abaqus value	MATLAB value	UoM
1	NT11 N:21	38.6765	40.00000	°C
2	NT11 N:23	57.6329	57.88403	°C
3	NT11 N:25	69.0987	69.15628	°C
4	NT11 N:39	21.5550	22.11597	°C
5	NT11 N:41	39.5222	40.00000	°C
6	NT11 N:43	57.6329	57.88403	°C
7	NT11 N:57	10.5401	10.84372	°C
8	NT11 N:59	21.5550	22.11597	°C
9	NT11 N:61	38.6765	40.00000	°C

Table 2.2: Nodal temperature results of Problem 2.

The results presented in Table 2.2 demonstrate that the temperature values obtained from Abaqus and MATLAB are remarkably similar, indicating that the BEM maintains high accuracy while offering significant computational efficiency advantages over the FEM.

Additionally, the results highlight a clear line of symmetry in the temperature distribution, which is consistent with the boundary conditions and the geometry of the problem:

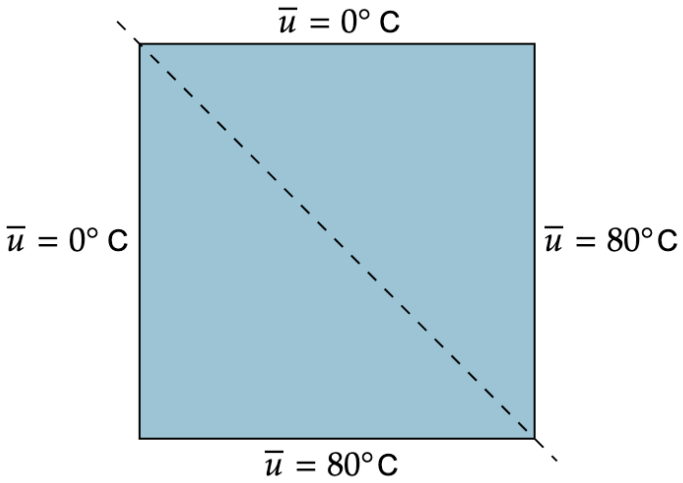


Figure 2.24: Simmetry with respect to the diagonal line.

2.6. Problem 3

In this final problem we search for the solution of a simple potential problem with mixed boundary conditions. The interesting thing is that the square domain Ω is doubly connected, namely, it contains a hole. Its outer boundary has been discretized into 16 constant elements, while the inner one into 8 elements. The solution is sought at 8 internal points. The data and the boundary discretization are shown in Fig. 2.25 and Fig. 2.26.

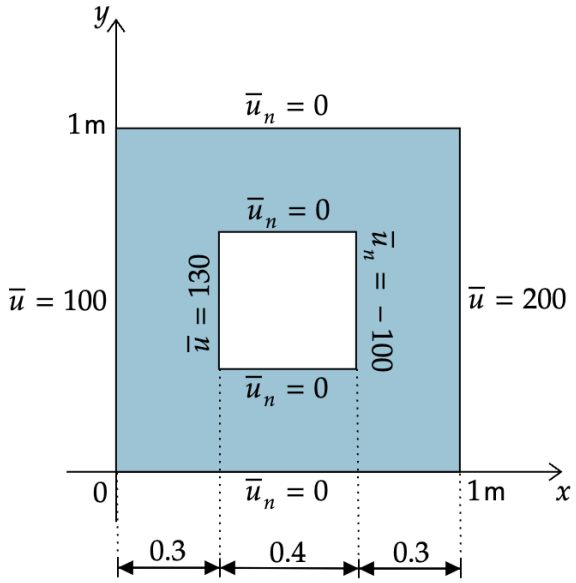


Figure 2.25: Doubly connected square domain Ω and boundary conditions of Problem 3.

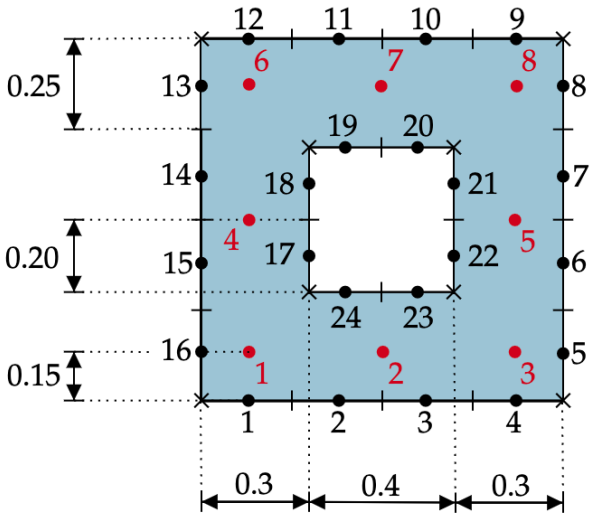


Figure 2.26: Boundary element discretization and internal points (in red) of Problem 3.

2.6.1. MATLAB Implementation

The program we used to solve Problem 1 and 2 can be readily modified to solve potential problems in multiply connected domains (domains with holes). The changes affect only the functions `MIDPOINTS` (see Algorithm 2.12), `GMATR` (see Algorithm 2.13), `HMATR` (see Algorithm 2.14), `UINTER` (see Algorithm 2.15) and of course the print functions, while the overall structure of the `main` script (see Algorithm 2.16) remains as before (see Eq. (2.23)). We only added two new parameters to the ones already presented in Table 2.1. One is NB and it defines the number of boundaries, the second is the vector NL(i) which identifies the number of the last element on the i-th boundary ($i = 1, \ldots, \text{NB}$). It should also be noted that the elements of all the boundaries are numbered consecutively and, therefore, N denotes the total number of elements.

The output of the program is the following:

```
**********************************************************************
RESULTS
**********************************************************************

BOUNDARY NODES:

NODE        XM            YM            U                U_n
 1        0.12500       0.00000    1.121172e+02          0.00000
 2        0.37500       0.00000    1.377572e+02          0.00000
 3        0.62500       0.00000    1.627310e+02          0.00000
 4        0.87500       0.00000    1.881075e+02          0.00000
 5        1.00000       0.12500             200        105.36171
 6        1.00000       0.37500             200         95.21982
 7        1.00000       0.62500             200         95.21982
 8        1.00000       0.87500             200        105.36171
 9        0.87500       1.00000    1.881075e+02          0.00000
 10       0.62500       1.00000    1.627310e+02          0.00000
 11       0.37500       1.00000    1.377572e+02          0.00000
 12       0.12500       1.00000    1.121172e+02          0.00000
 13       0.00000       0.87500             100       -107.76583
 14       0.00000       0.62500             100        -99.55986
 15       0.00000       0.37500             100        -99.55986
 16       0.00000       0.12500             100       -107.76583
----------------------------------------------------------------------
 17       0.30000       0.40000             130        108.32831
 18       0.30000       0.60000             130        108.32831
 19       0.40000       0.70000    1.405389e+02          0.00000
 20       0.60000       0.70000    1.600210e+02          0.00000
 21       0.70000       0.60000    1.716784e+02       -100.00000
 22       0.70000       0.40000    1.716784e+02       -100.00000
 23       0.60000       0.30000    1.600210e+02          0.00000
 24       0.40000       0.30000    1.405389e+02          0.00000
```

```
 INTERNAL POINTS:

 POINT        XIN           YIN           U
   1        0.15000       0.15000     115.08022
   2        0.50000       0.15000     150.26437
   3        0.85000       0.15000     185.25460
   4        0.15000       0.50000     115.07475
   5        0.85000       0.50000     185.68733
   6        0.15000       0.85000     115.08022
   7        0.50000       0.85000     150.26437
   8        0.85000       0.85000     185.25460
```

2.6.2. MATLAB scripts

Algorithm 2.12: holeMIDPOINTS

```
function [XM, YM] = holeMIDPOINTS(XL, YL, NB, NL)
XM = zeros(NL(end),1);
YM = zeros(NL(end),1);
beg = 1;
for i = 1:NB
    for j = beg:NL(i)
        if j==NL(i)
            XM(j) = (XL(j)+XL(beg))/2;
            YM(j) = (YL(j)+YL(beg))/2;
        else
            XM(j) = (XL(j)+XL(j+1))/2;
            YM(j) = (YL(j)+YL(j+1))/2;
        end
    end
    beg = NL(i)+1;
end
end
```

Algorithm 2.13: holeGMATR

```
function G = holeGMATR(XL, YL, XM, YM, N, NB, NL)
G = zeros(N,N); beg = 1;
for k = 1:NB
    for i = 1:N
        for j = beg:NL(k)
            if i~=j % off-diagonal elements of G
                if j==NL(k)
                    G(i,j) = RLINTC(XM(i),YM(i),XL(j),YL(j),XL(beg),YL(
   beg));
```

```
                else
                    G(i,j) = RLINTC(XM(i),YM(i),XL(j),YL(j),XL(j+1),YL(j
   +1));
                end
            elseif i==j % diagonal elements of G
                if j==NL(k)
                    G(i,j) = SLINTC(XL(j),YL(j),XL(beg),YL(beg));
                else
                    G(i,j) = SLINTC(XL(j),YL(j),XL(j+1),YL(j+1));
                end
            end
        end
    end
    beg = NL(k)+1;
end
end
```

Algorithm 2.14: holeHMATR

```
function H = holeHMATR(XL, YL, XM, YM, N, NB, NL)
H = zeros(N,N);
beg = 1;
for k=1:NB
    for i = 1:N
        for j = beg:NL(k)
            if i~=j % off-diagonal elements of G
                if j==NL(k)
                    H(i,j) = DALPHA(XM(i),YM(i),XL(j),YL(j),XL(beg),YL(
   beg));
                else
                    H(i,j) = DALPHA(XM(i),YM(i),XL(j),YL(j),XL(j+1),YL(j
   +1));
                end
            elseif i == j % diagonal elements of G
                H(i,j) = -0.5;
            end
        end
    end
    beg = NL(k)+1;
end
end
```

Algorithm 2.15: holeUINTER

```
function UIN = holeUINTER(XL, YL, XIN, YIN, UB, UNB, N, IN, NL, NB)
UIN = zeros(IN, 1);
for k = 1:IN
    for j = 1:N
        if NB == 1
            resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(j+1), YL(j+1))
   ;
            resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j), XL(j+1), YL(j+1))
   ;
            UIN(k) = UIN(k) + resh*UB(j) - resg*UNB(j);
        else
            if j == NL(1)
                resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(1), YL(1))
   ;
                resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j), XL(1), YL(1))
   ;
                UIN(k) = UIN(k) + resh*UB(j) - resg*UNB(j);
            else
                for kk = 2:NB
                    if j == NL(kk)
                        resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(
   NL(kk-1)+1  ), YL( NL(kk-1)+1 ));
                        resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j),  XL(
   NL(kk-1)+1  ), YL( NL(kk-1)+1 ));
                       UIN(k) = UIN(k) + resh*UB(j) - resg*UNB(j);
                    else
                        resh = DALPHA(XIN(k), YIN(k), XL(j), YL(j), XL(j
   +1), YL(j+1));
                        resg = RLINTC(XIN(k), YIN(k), XL(j), YL(j), XL(j
   +1), YL(j+1));
                        UIN(k) = UIN(k) + resh*UB(j) - resg*UNB(j);
                    end
                end
            end
        end

    end
end
end
```

Algorithm 2.16: main23

```
%% Problem 2 - Square hole in a square
clear; clc;
```

```
% Set the maximum dimensions
N = 24;
IN = 8;
NB = 2; % # boundaries
NL = [16; 24]; % last point of each boundary

% Set data
XL = [0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25; 0; 0; 0; 0;
      0.3; 0.3; 0.3; 0.5; 0.7; 0.7; 0.7; 0.5];
YL = [0; 0; 0; 0; 0; 0.25; 0.5; 0.75; 1; 1; 1; 1; 1; 0.75; 0.5; 0.25;
      0.3; 0.5; 0.7; 0.7; 0.7; 0.5; 0.3; 0.3];
INDEX = [1; 1; 1; 1; 0; 0; 0; 0; 1; 1; 1; 1; 0; 0; 0; 0;
         0; 0; 1; 1; 1; 1; 1; 1];
UB = [0; 0; 0; 0; 200; 200; 200; 200; 0; 0; 0; 0; 100; 100; 100; 100;
      130; 130; 0; 0; -100; -100; 0; 0];
XIN = [0.15; 0.5; 0.85; 0.15; 0.85; 0.15; 0.5; 0.85];
YIN = [0.15; 0.15; 0.15; 0.5; 0.5; 0.85; 0.85; 0.85];

% Compute midpoints
[XM, YM] = holeMIDPOINTS(XL, YL, NB, NL);

% Print data
holeINPUT(XL, YL, XM, YM, XIN, YIN, INDEX, UB, N, IN, NB, NL);

% % Compute the G matrix
G = holeGMATR(XL, YL, XM, YM, N, NB, NL);

% Compute the H matrix
H = holeHMATR(XL, YL, XM, YM, N, NB, NL);

% Form the system of equations AX=B
[A, UNB] = ABMATR(G, H, UB, INDEX);

% Solve the system of equations
UNB = A\UNB;

% Form the vectors U and UN of all the boundary values
[UB, UNB] = REORDER(UB, UNB, INDEX);

% Compute the values UIN of u at the internal points
UIN = holeUINTER(XL, YL, XIN, YIN, UB, UNB, N, IN, NL, NB);

% Print results
holeOUTPUT(XM, YM, XIN, YIN, UB, UNB, UIN, N, IN, NL, NB);
```

2.6.3. Abaqus Implementation

The Abaqus procedure is symmetric to the ones of Problem 2, and described in Section 2.5.3. Let us mention the main passages:

- Part module: we create a 2D planar deformable shell part of size 2000, then we create the square with a hole.

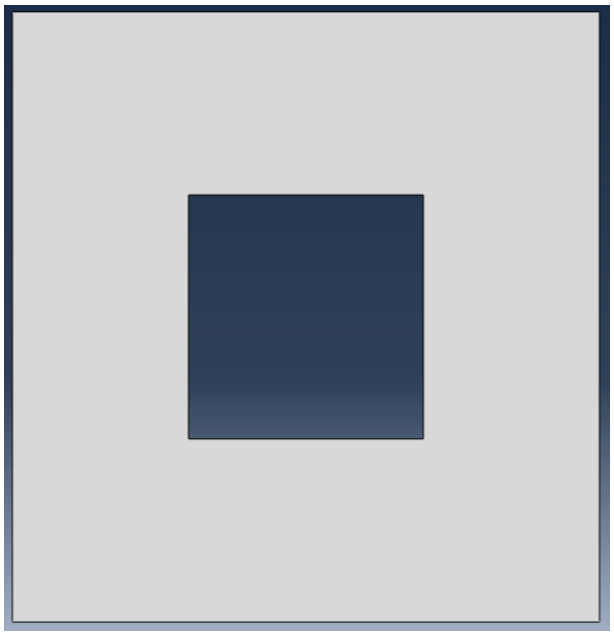


Figure 2.27: Final result of Part module.

- Property module: we define a material with conductivity $k = 1000$ (see Eq. (2.24)) and we assign it to a solid homogeneous section.
- Assemply module: we create a dependent instance.
- Step module: we define a steady state heat transfer step.
- Load module: we assign the boundary conditions.

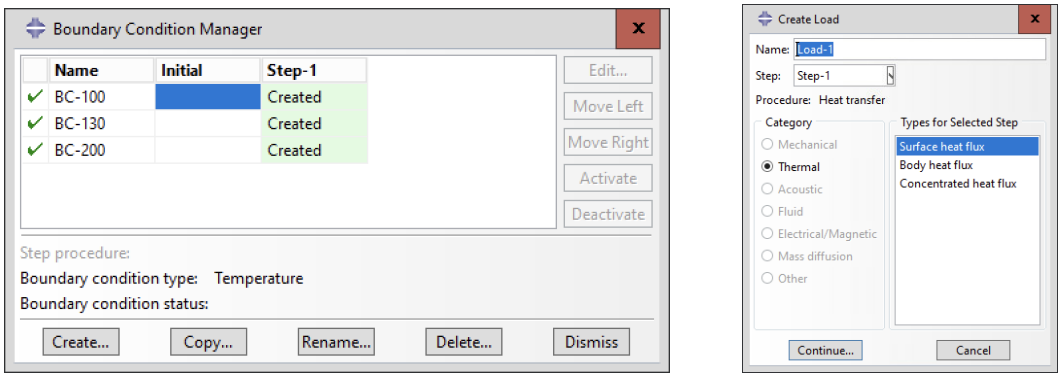


Figure 2.28: Boundary conditions.

This is what we obtained so far:

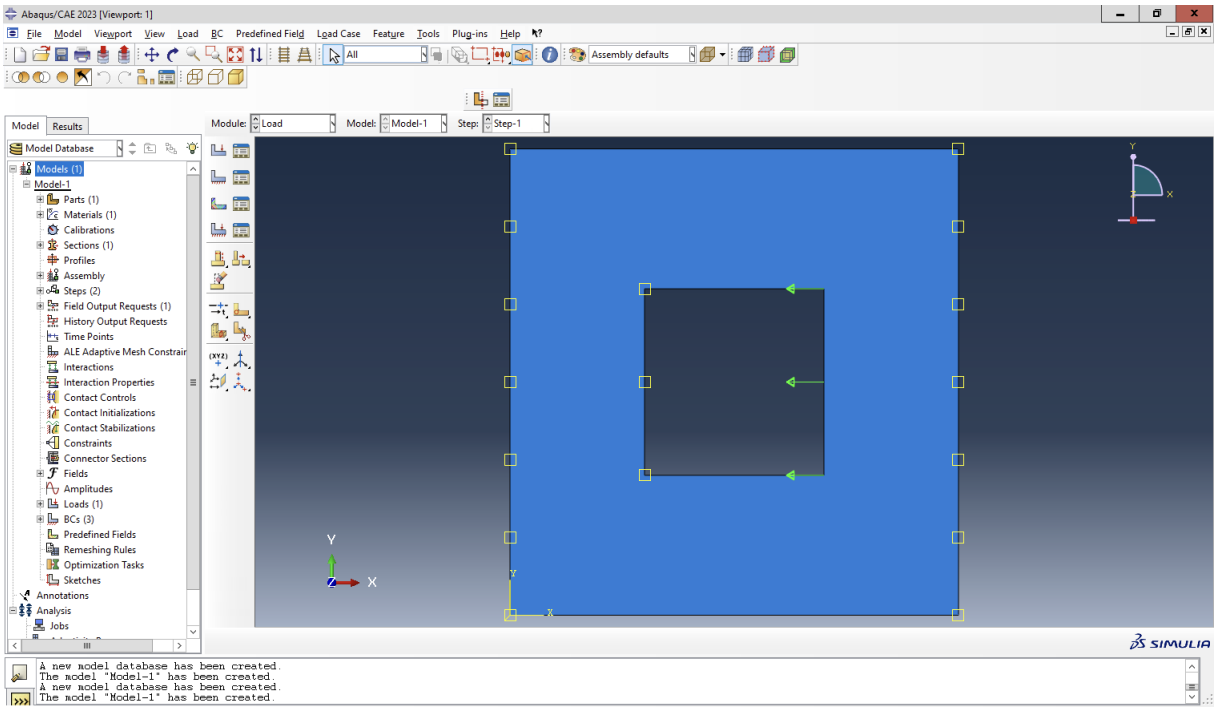


Figure 2.29: Final result of Load module.

- Mesh module: we assign the heat transfer element type, then in Mesh Controls we set the quad shape and the medial axis algorithm.

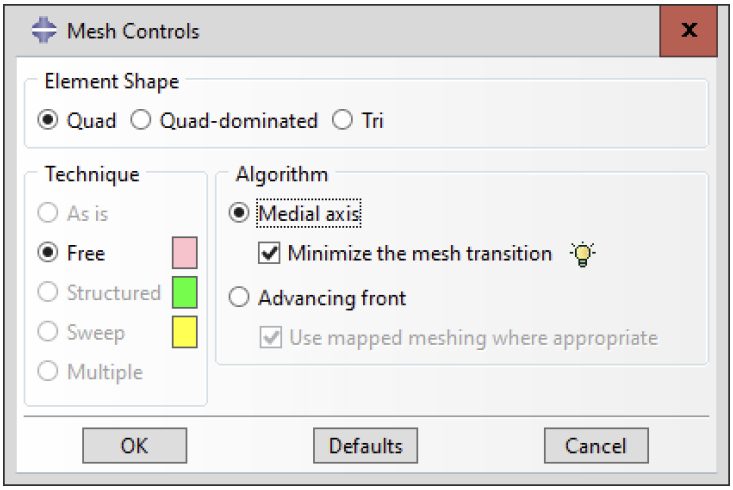


Figure 2.30: Mesh Controls.

In the Global Seeds option, we define a length of 50 mm for the quads edges: $1000\,\text{mm}/50\,\text{mm} = 20$ global size.

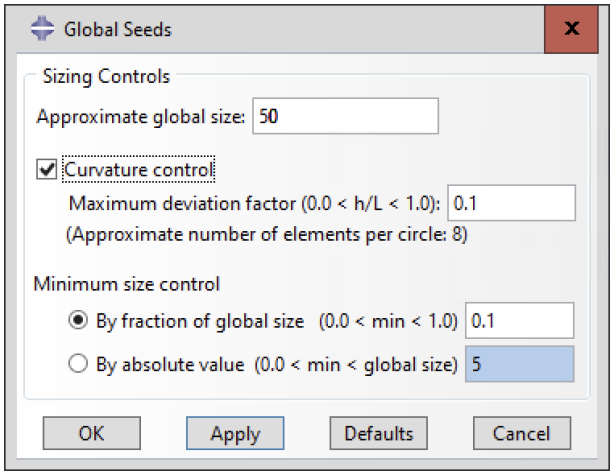


Figure 2.31: Global Seeds.

Here the final mesh:

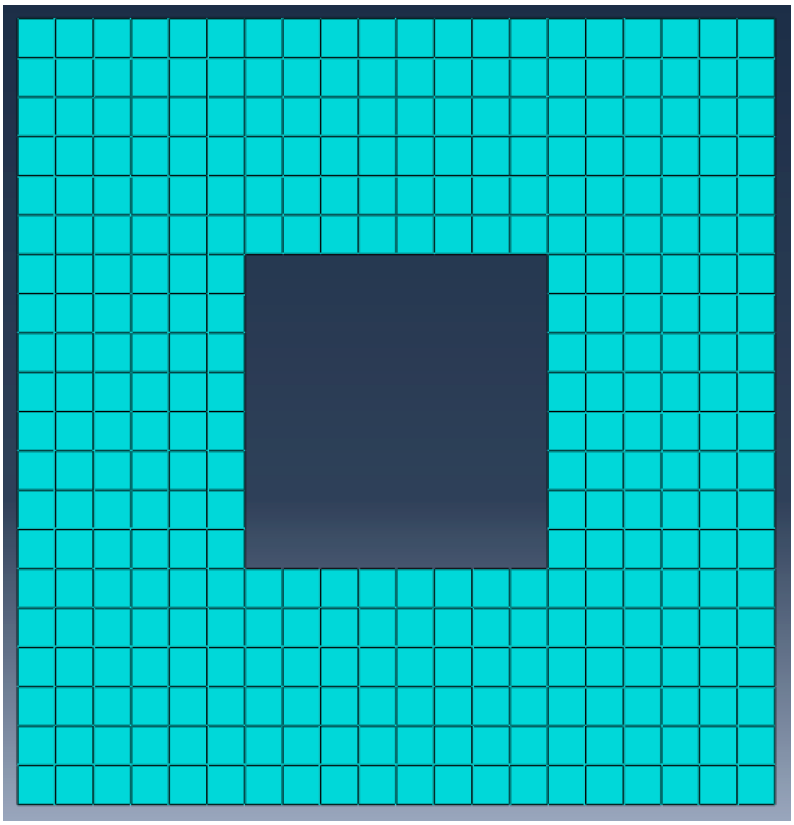


Figure 2.32: Final result of Mesh module.

- Job module: finaly, we submit a standard job.

In the Visualization module, we observe the nodal temperature distributions:

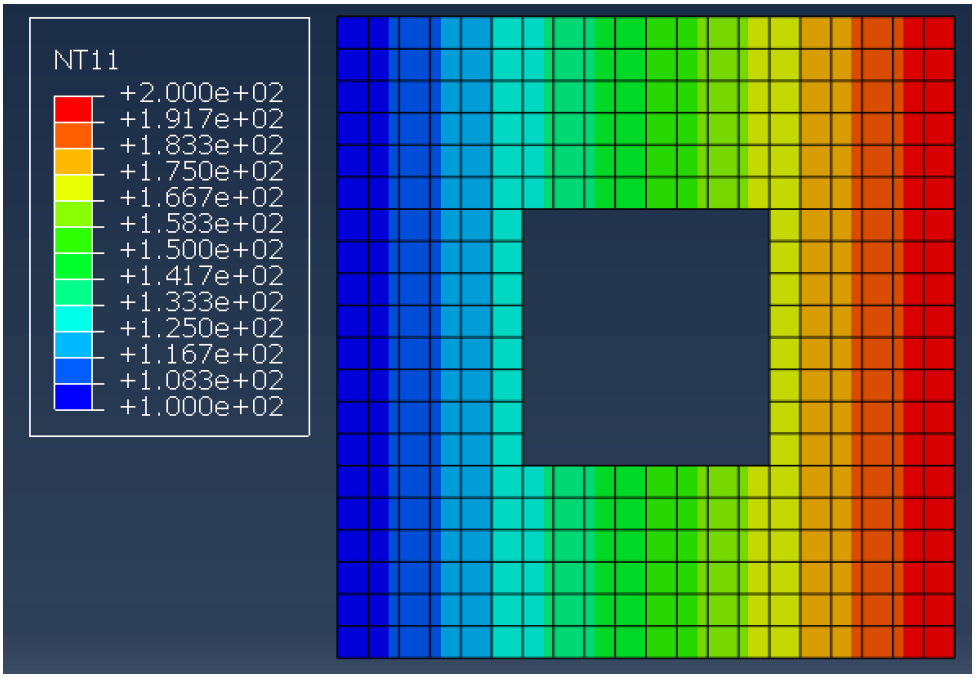
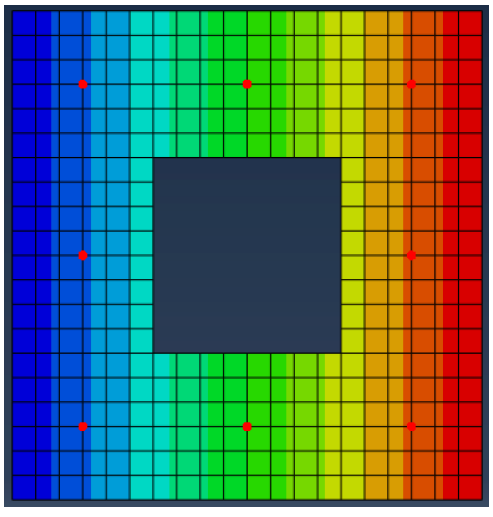


Figure 2.33: Nodal temperature.

Then, we extract the numerical values of the eight internal points:



Part Instance	Node ID	Orig. Coords	Def. Coords	Attached elements	NT11
PART-1-1	280	150, 150, 0	150, 150, 0	183, 184, 189, 190	115
PART-1-1	245	500, 150, 0	500, 150, 0	141, 142, 147, 148	150
PART-1-1	145	850, 150, 0	850, 150, 0	15, 16, 21, 22	185
PART-1-1	375	150, 500, 0	150, 500, 0	309, 310, 315, 316	115
PART-1-1	180	850, 500, 0	850, 500, 0	57, 58, 63, 64	185
PART-1-1	340	150, 850, 0	150, 850, 0	267, 268, 273, 274	115
PART-1-1	310	500, 850, 0	500, 850, 0	225, 226, 231, 232	150
PART-1-1	215	850, 850, 0	850, 850, 0	99, 100, 105, 106	185

Figure 2.34: Nodal temperature of internal points.

The numerical values are tabulated in the next section.

2.6.4. Comparison between MATLAB and Abaqus

Here we present the comparison of the results obtained from solving Problem 3 using MATLAB and Abaqus software. The comparison focuses on the temperature distribution at eight selected internal nodes, as depicted in Fig. 2.26.

Internal Point label	Abaqus label	Abaqus value	MATLAB value	UoM
1	NT11 N:280	115	115.08022	°C
2	NT11 N:245	150	150.26437	°C
3	NT11 N:145	185	185.25460	°C
4	NT11 N:375	115	115.07475	°C
5	NT11 N:180	185	185.68733	°C
6	NT11 N:340	115	115.08022	°C
7	NT11 N:310	150	150.26347	°C
8	NT11 N:215	185	185.25460	°C

Table 2.3: Nodal temperature results of Problem 3.

The results presented in Table 2.2 demonstrate that the temperature values obtained from Abaqus and MATLAB are remarkably similar.

In addiction, these results perfectly match the ones obtained with FORTRAN in [1], and they can be compared with the exact solution $u(x, y) = 100(1 + x)$, showing a rapid convergence.

Bibliography

[1] J. T. Katsikadelis. *The Boundary Element Method for Engineers and Scientist*. Academic Press, 2 edition, 2016.

[2] N. H. Kim. *Introduction to Nonlinear Finite Element Analysis*. Springer, 2015.

[3] Y. J. Liu, S. Mukherjee, N. Nishimura, M. Schanz, W. Ye, A. Sutradhar, E. Pan, A. Dumont, A. Frangi, and A. Saez. Recent advances and emerging applications of the boundary element method. *Applied Mechanics Reviews*, (64), 2011.

[4] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer, 2007.