Inhomogeneous Helmholtz PDE solutions inside an L-Shaped Domain

Project in Advanced Scientific Computing

Writer: Zikos Ioannis, 58906

Professors: George A. Gravvanis, Christos K. Filelis-Papadopoulos

June 2025

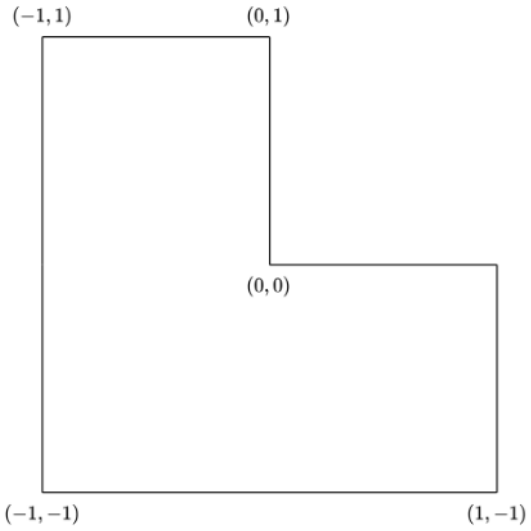
Contents

1. The General Problem

Given the following PDE:

$$\Delta u + k^2 * u(x,y) = -f(x,y), (x,y) \in \Omega$$
$$u(x,y) = 0, (x,y) \in \partial\Omega$$

Where **Ω** is the domain in which the above PDE is defined and is of the following format:



And f(x, y) is:

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x+0.5)^2 + (y-0.5)^2}{2\sigma^2}} + \frac{1}{2\pi\sigma^2} e^{-\frac{(x-0.5)^2 + (y+0.5)^2}{2\sigma^2}}, (x,y \in \Omega)$$

$$with\ \sigma = 0.05$$

In the next chapters, we will construct the square mesh of the domain, compute the Stiffness Matrix and the RHS (Right Hand Side) of the problem based on the FEM (Finite Elements Method). Then we will proceed to solve the PDE using various computational methods like Sparse LU Decomposition, Conjugate Gradient, Domain Decomposition and preconditioned versions of the Conjugate Gradient in the Julia programming language. All the code blocks and functions in this project will be written in Julia (.jl file type) even if it is not discreetly mentioned.

2. Square Meshing the L-Shaped Domain

This L-Shaped domain of our problem can be discretized into a square mesh, with given max square width, using the Julia function below:

Function 1. *Lmesh.jl*

```
"""
    Lmesh(maxSQlen::Float64)

# Arguments
 - maxSQlen::Float64 : The maximum side size of the square with which you want to create the L mesh. (This is the max length, if it doesn't divide the domain equally the function will take nearest value below maxSQlen which creates an even mesh)

# Returns
- go : Global Ordering (go) is a matrix with 3 other martices (the subdomains) inside it, those 3 matrices contain the mesh of each subdomain
"""
function Lmesh(maxSQlen::Float64)

    domain = [-1 0 1;] # Domain

    # Func input
    domStep = abs(domain[2] - domain[1]);
    maxSQlen = min(maxSQlen, domStep); # Maximum square length (not      greater than 1 so that we dont overwrite the domain)
    maxSQ = div(domStep, maxSQlen, RoundUp); # Maximum small squares inside the big square based on maxSQlen
    step1 =  domStep/maxSQ; # Final closest step so that all small squares have equal length <= maxSQlen

    #Domain properties
    n = length(domain);
    domMid = (n+1)/2;

    # ---- Square 1 & 2 Mesh --------------------------------

    x1 = range(domain[1], domain[Integer(domMid)], step = step1) |> collect; # Nodes vector in row 1
    rowNodes = length(x1);
    rowElem = length(x1) - 1; # Elements in 1 row

    # Global ordering of Square 1 & 2

    go1 = (rowNodes-1:-1:0).*(2*rowNodes-1).+(1:rowNodes)'; # Only use matrix operations to create the mesh
```

```
    lastNode = go1[end,end]; # Temp to hold the last node so that we can
construct the next matrix

    go2 = (rowNodes-1:-1:0).*(2*rowNodes-1).+(lastNode:lastNode+rowNodes-1)';
    lastNode = go2[1,end];

    # ---- Square 3 Mesh -------------------------------

    go3 = zeros(Int, rowElem+1, rowElem+1);
    go3[end,:] = go1[1,:];
    go3[1:end-1, :] = (rowNodes-2:-
1:0).*(rowNodes).+(lastNode+1:lastNode+rowNodes)';

    go = [go1, go2, go3]; # Use go[1] to get go1, go[2] to get go2 etc.

    return go
end
```

This function takes as input a Float64 named maxSQlen which represents the maximum width of the square. Then using the geometry of the domain, it sets the width of the square (also referred to as step) equal to the input or the closest possible value to the input, without surpassing it, that divides the domain into equal sized squares. This is accomplished by dividing, conceptually, the domain into 3 square subdomains and then using vectors and the final step, that was decided earlier, to create the Global Ordering of the domain following the natural ordering (starting the index from the bottom left corner and then continuing up from left to right). Finally, this function outputs 3 matrices, one for each subdomain, which together they give us all the nodes inside the global L-shaped domain, effectively meshing it. The meshed domain, for different steps, is displayed in the Figures below (we used the function PlotDomain.jl, look at *Appendix A*.):

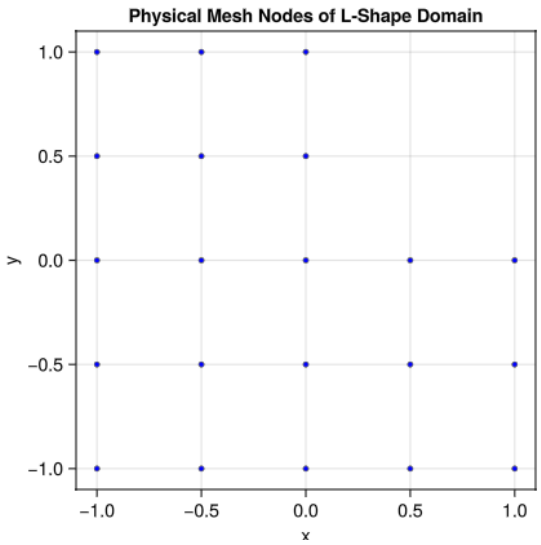


Figure 1. *Physical Mesh Nodes of L-Shaped Domain for step = 0.5*

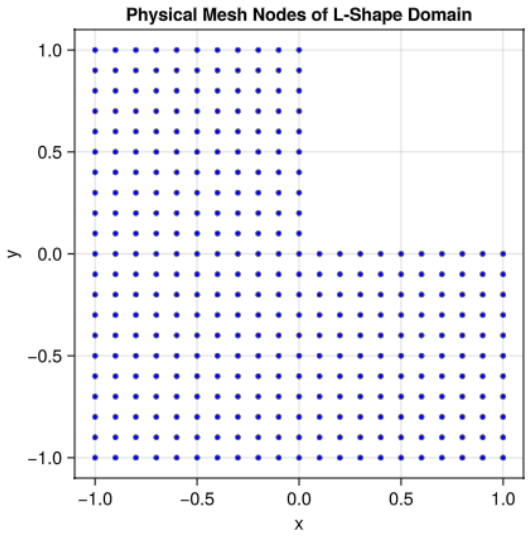


Figure 2. *Physical Mesh Nodes of L-Shaped Domain for step = 0.1*

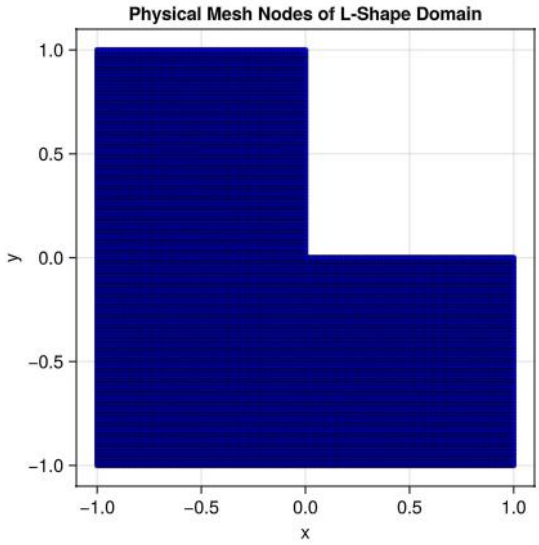


Figure 3. *Physical Mesh Nodes of L-Shaped Domain for step = 0.01*

3. Computing FEM Stiffness Matrix and RHS

A quadrilateral element has four local nodes which are numbered in a counter-clockwise direction. Knowing the solution at the four nodes of the element, the primary unknown quantity can be evaluated at any point inside the element by using the appropriate interpolation functions. That is what we, amongst others, will calculate in this section. The master element, which is defined in the ξ η-coordinate system (natural coordinate system) has the square shape shown below:

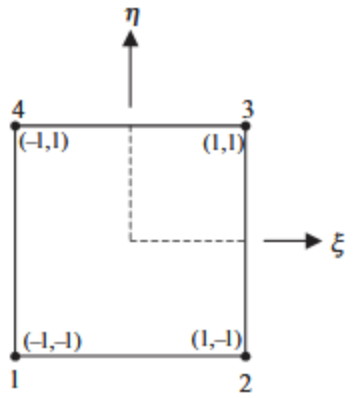


Figure 4. *Quadrilateral master element in the ξ η-plane*

A generic interpolation function for local node 1 spanning the geometrical domain of the master quadrilateral element has the form:

$$N_1(\xi, \eta) = c_1 + c_2\xi + c_3\eta + c_4\xi\eta \qquad (1)$$

And according to the properties of the Lagrange polynomials:

$$N_1 = \begin{cases} 1 \; at \; node \; 1 \\ 0 \; at \; all \; other \; nodes \end{cases}$$

Applying these conditions at the four nodes of the master quadrilateral element, the result is a system of four equations with four unknowns, the unknowns being the four constants of (1).

$$N_1(-1,-1) = c_1 - c_2 - c_3 + c_4 = 1$$
$$N_1(1,-1) = c_1 + c_2 - c_3 - c_4 = 0$$
$$N_1(1,1) = c_1 + c_2 + c_3 + c_4 = 0$$
$$N_1(-1,1) = c_1 - c_2 + c_3 - c_4 = 0$$

After solving the system for each constant, we arrive at:

$$N_1(\xi,\eta) = \frac{1}{4}(1-\xi)(1-\eta)$$

Similarly, we construct all four interpolation functions:

$$\begin{aligned} N_1(\xi,\eta) &= \frac{1}{4}(1-\xi)(1-\eta) \\ N_2(\xi,\eta) &= \frac{1}{4}(1+\xi)(1-\eta) \\ N_3(\xi,\eta) &= \frac{1}{4}(1+\xi)(1+\eta) \\ N_4(\xi,\eta) &= \frac{1}{4}(1-\xi)(1+\eta) \end{aligned} \tag{2}$$

Assuming an isoparametric quadrilateral element, the primary unknown quantity and the x and y space coordinates inside an element can be expressed in terms of these four basis functions given by (2)

$$u = u_1^e N_1 + u_2^e N_2 + u_3^e N_3 + u_4^e N_4 = \sum_{i=1}^{4} u_i^e N_i \tag{4}$$

And

$$\begin{aligned} x &= x_1^e N_1 + x_2^e N_2 + x_3^e N_3 + x_4^e N_4 = \sum_{i=1}^{4} x_i^e N_i \\ y &= y_1^e N_1 + y_2^e N_2 + y_3^e N_3 + y_4^e N_4 = \sum_{i=1}^{4} y_i^e N_i \end{aligned} \tag{5}$$

The Partial Differential Equation of our problem can also be expressed as:

$$\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}\right) + k^2 * u = -f \tag{6}$$

The weak formulation of this problem can be obtained by first constructing the weighted residual of the equation above for a single element with domain Ω^e. The element residual is formed by moving the right-hand side to the left-hand side:

$$r^e = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}\right) + \mathrm{k}^2 * u + f$$

This element residual is ideally zero, provided that the numerical solution u to be obtained is identical to the exact solution. To minimize this element residual in a weighted sense we must first multiply r^e with a weight function w, then integrate the result over the area of the

element, and finally, set the integral to zero:

$$\iint_{\Omega_e} w\left[\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}\right)+k^2u+f\right]dxdy=0 \tag{7}$$

Where because:

$$w\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)=\frac{\partial}{\partial x}\left(w\frac{\partial u}{\partial x}\right)-\frac{\partial w}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

We get this from substituting the latter into the integral (7):

$$\begin{aligned}\iint_{\Omega_e}\left[\frac{\partial}{\partial x}\left(w\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(w\frac{\partial u}{\partial y}\right)\right]dx\,dy-\iint_{\Omega_e}\left[\frac{\partial w}{\partial x}\frac{\partial u}{\partial x}+\frac{\partial w}{\partial y}\frac{\partial u}{\partial y}\right]dx\,dy\\ +\iint_{\Omega_e}k^2w\,udxdy=-\iint_{\Omega_e}wf\,dxdy\end{aligned} \tag{8}$$

Applying Green's theorem to the first integral of (8):

$$\iint_{\Omega^e}\left[\frac{\partial}{\partial x}\left(w\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(w\frac{\partial u}{\partial y}\right)\right]dx\,dy=\oint_{\Gamma^e}w\left(\frac{\partial u}{\partial x}n_x+\frac{\partial u}{\partial y}n_y\right)dl$$

We substitute this result to (8) and the weak form of the differential equation reduces to:

$$\begin{aligned}-\iint_{\Omega^e}\left[\frac{\partial w}{\partial x}\frac{\partial u}{\partial x}+\frac{\partial w}{\partial y}\frac{\partial u}{\partial y}\right]dx\,dy+\iint_{\Omega^e}k^2w\,udxdy\\ =-\iint_{\Omega^e}wf\,dxdy-\oint_{\Gamma^e}w\left(\frac{\partial u}{\partial x}n_x+\frac{\partial u}{\partial y}n_y\right)dl\end{aligned} \tag{9}$$

Now substituting (4) into (9) and setting:

$$w=N_i\ for\ i\ =\ 1,2,\ldots,n$$

according to the Galerkin approach, the weak form of our PDE now becomes:

$$\begin{aligned}&-\iint_{\Omega^e}\left[\frac{\partial N_i}{\partial x}\frac{\partial\left(\sum_{j=1}^{n}u_j^eN_j\right)}{\partial x}+\frac{\partial N_i}{\partial y}\frac{\partial\left(\sum_{j=1}^{n}u_j^eN_j\right)}{\partial y}\right]dxdy\\ &+\iint_{\Omega^e}k^2N_i\left(\sum_{=1}^{n}u_j^eN_j\right)dxdy=-\iint_{\Omega^e}N_ifdxdy-\oint_{\Gamma^e}N_i\left(\frac{\partial u}{\partial x}n_x+\frac{\partial u}{\partial y}n_y\right)dl\\ &for\ i=1,2,\ldots,n\end{aligned} \tag{10}$$

Equation (10) can also be written in the following form:

$$\begin{bmatrix} M_{11}^e & M_{12}^e & \cdots & M_{1n}^e \\ M_{21}^e & M_{22}^e & \cdots & M_{2n}^e \\ \vdots & \vdots & \ddots & \vdots \\ M_{n1}^e & M_{n2}^e & \cdots & M_{nn}^e \end{bmatrix} \begin{Bmatrix} u_1^e \\ u_2^e \\ \vdots \\ u_n^e \end{Bmatrix} + \begin{bmatrix} T_{11}^e & T_{12}^e & \cdots & T_{1n}^e \\ T_{21}^e & T_{22}^e & \cdots & T_{2n}^e \\ \vdots & \vdots & \ddots & \vdots \\ T_{n1}^e & T_{n2}^e & \cdots & T_{nn}^e \end{bmatrix} \begin{Bmatrix} u_1^e \\ u_2^e \\ \vdots \\ u_n^e \end{Bmatrix} = \begin{Bmatrix} f_1^e \\ f_2^e \\ \vdots \\ f_n^e \end{Bmatrix} + \begin{Bmatrix} p_1^e \\ p_2^e \\ \vdots \\ p_n^e \end{Bmatrix} \quad (11)$$

Where:

$$M_{ij}^e = -\iint_{\Omega^e} \left[\left(\frac{\partial N_i}{\partial x} \right) \left(\frac{\partial N_j}{\partial x} \right) + \left(\frac{\partial N_i}{\partial y} \right) \left(\frac{\partial N_j}{\partial y} \right) \right] dx\, dy \quad (12)$$

$$T_{ij}^e = \iint_{\Omega^e} k^2 N_i N_j dx\, dy \quad (13)$$

$$f_i^e = -\iint_{\Omega^e} N_i f\, dxdy \quad (14)$$

$$p_i^e = -\oint_{\Gamma^e} N_i \left(\frac{\partial u}{\partial x} n_x + \frac{\partial u}{\partial y} n_y \right) dl \quad (15)$$

In a more compact form the matrix system from above becomes:

$$\begin{bmatrix} K_{11}^e & K_{12}^e & \cdots & K_{1n}^e \\ K_{21}^e & K_{22}^e & \cdots & K_{2n}^e \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1}^e & K_{n2}^e & \cdots & K_{nn}^e \end{bmatrix} \begin{Bmatrix} u_1^e \\ u_2^e \\ \vdots \\ u_n^e \end{Bmatrix} = \begin{Bmatrix} b_1^e \\ b_2^e \\ \vdots \\ b_n^e \end{Bmatrix}$$

Where:

$$K_{ij}^e = M_{ij}^e + T_{ij}^e$$
$$b_i^e = f_i^e + p_i^e$$

.

Here, K_{ij}^e is our Stiffness Matrix and b_i^e is our RHS (Right-Hand Side)

The partial derivatives of the interpolation functions in (2) with respect to ξ and η can be expressed as:

$$\frac{\partial N_i}{\partial \xi} = \frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \xi}$$
$$\frac{\partial N_i}{\partial \eta} = \frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \eta} + \frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \eta} \quad (16)$$

Or

$$\begin{Bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{Bmatrix}$$
$$= J \begin{Bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{Bmatrix}$$

Where J is the Jacobian matrix:

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix}$$

With

$$\begin{aligned} J_{11} &= \frac{1}{4}[-(1-\eta)x_1^e + (1-\eta)x_2^e + (1+\eta)x_3^e - (1+\eta)x_4^e] \\ J_{12} &= \frac{1}{4}[-(1-\eta)y_1^e + (1-\eta)y_2^e + (1+\eta)y_3^e - (1+\eta)y_4^e] \\ J_{21} &= \frac{1}{4}[-(1-\xi)x_1^e - (1+\xi)x_2^e + (1+\xi)x_3^e + (1-\xi)x_4^e] \\ J_{22} &= \frac{1}{4}[-(1-\xi)y_1^e - (1+\xi)y_2^e + (1+\xi)y_3^e + (1-\xi)y_4^e] \end{aligned} \tag{17}$$

And

$$|J| = J_{11}J_{22} - J_{12}J_{21} \tag{18}$$

Combining these we get that:

$$\begin{cases} \frac{\partial N_1}{\partial x} = \frac{1}{4|J|}[-J_{22}(1-\eta) + J_{12}(1-\xi)] \\ \frac{\partial N_1}{\partial y} = \frac{1}{4|J|}[J_{21}(1-\eta) - J_{11}(1-\xi)] \end{cases} \tag{19}$$

$$\begin{cases} \frac{\partial N_2}{\partial x} = \frac{1}{4|J|}[J_{22}(1-\eta) + J_{12}(1+\xi)] \\ \frac{\partial N_2}{\partial y} = \frac{1}{4|J|}[-J_{21}(1-\eta) - J_{11}(1+\xi)] \end{cases} \tag{20}$$

$$\begin{cases} \dfrac{\partial N_3}{\partial x} = \dfrac{1}{4|J|}[J_{22}(1+\eta) - J_{12}(1+\xi)] \\ \dfrac{\partial N_3}{\partial y} = \dfrac{1}{4|J|}[-J_{21}(1+\eta) + J_{11}(1+\xi)] \end{cases} \tag{21}$$

$$\begin{cases} \dfrac{\partial N_4}{\partial x} = \dfrac{1}{4|J|}[-J_{22}(1+\eta) - J_{12}(1-\xi)] \\ \dfrac{\partial N_4}{\partial y} = \dfrac{1}{4|J|}[J_{21}(1+\eta) + J_{11}(1-\xi)] \end{cases} \tag{22}$$

Now using the Jacobi transformation, for our quadrilateral elements, the integral from (12) can be written as:

$$M_{ij}^e = -\int_{-1}^{1}\int_{-1}^{1}\left[\left(\frac{\partial N_i}{\partial x}\right)\left(\frac{\partial N_j}{\partial x}\right) + \left(\frac{\partial N_i}{\partial y}\right)\left(\frac{\partial N_j}{\partial y}\right)\right]|J|d\xi d\eta \tag{23}$$

And similarly, we can transform (13) and (14) to:

$$T_{ij}^e = \int_{-1}^{1}\int_{-1}^{1} k^2 N_i N_j |J| d\xi d\eta \tag{24}$$

$$f_i^e = -\int_{-1}^{1}\int_{-1}^{1} N_i f |J| d\xi d\eta \tag{25}$$

These integrals now in (23), (24) and (25) can be evaluated with the Legendre Gaussian Quadrature technique.

Also, we have the right-hand side term p_i^e (15) which is going to be 0 at the common sides from element to element, because they cancel each other out, and is also going to be 0 at the boundaries of the domain since we have the condition: $u(x,y) = 0, (x,y) \in \partial\Omega$. So ultimately, the term p_i^e becomes 0 and we can ignore it.

Now we are ready to compute the Stiffness Matrix and the RHS. The Julia code below does exactly that:

Function 2. *QuadFEM_Matrices.jl*

```
"""
    QuadFEM_Matrices(GO::Vector{Matrix{Int64}}, f::Function, k::Float64)

# Arguments
- GO::Vector{Matrix{Int64}} : The Global Ordering matrix which contains the 3 seperate
subdomains matrices meshed (use Lmesh to create it)
- f::Function : The right hand side function of the PDE
- k::Float64 : k is the wave number (The bigger it is we expect more peaks to appear)
```

```
# Returns
- K : K is the sparse stiffness matrix of our inhomogeneous Helmholtz equation
- F_global : F_global is the final right hand side of the equation. Now use a solver to solve
the system K*u=F_global for u(x,y,t)
"""
function QuadFEM_Matrices(GO::Vector{Matrix{Int64}}, f::Function, k::Float64)

    rowNodes = length(GO[1][1,:]);
    colNodes = length(GO[1][:,1]);
    nop = 3*rowNodes^2 - 2*rowNodes; # Number of points
    rowElem = rowNodes - 1;
    colElem = colNodes - 1;
    noe = rowElem*colElem*3; # Number of elements
    step1 = 1/(rowNodes - 1); # Get step from the GO matrix

    # ---- Create local 2 global mapping ----------------------------------

    l2g = zeros(Int, noe, 4);

    idx = 1; # Index
    for i = 0:rowElem-1
        for j = 1:colElem
            l2g[idx, :] = [GO[1][end-i, j], GO[1][end-i, j+1], GO[1][end-i-1, j+1], GO[1][end-
i-1, j]];
            l2g[idx + Int(noe/3), :] = [GO[2][end-i, j], GO[2][end-i, j+1], GO[2][end-i-1,
j+1], GO[2][end-i-1, j]];
            l2g[idx + Int(2*noe/3), :] = [GO[3][end-i, j], GO[3][end-i, j+1], GO[3][end-i-1,
j+1], GO[3][end-i-1, j]];
            idx = idx + 1;
        end
    end

    # ---- Point Coords ---------------------------------------------------

    coords = zeros(nop,2);

    # Coordinates of points inside GO[1] & GO[2]
    for i = 1:rowNodes
        xvals = collect(-1:step1:1); # x values from -1 to 1 with step1
        yvals = (-1 + step1*(i - 1))*ones(rowNodes*2-1,1); # rowNodes*2-1 cause there is 1
common point, increase from -1 with step1 once every iteration

        start_idx = (i - 1)*(rowNodes*2-1) + 1;
        end_idx = (rowNodes*2-1)*i;
        coords[start_idx:end_idx, :] = hcat(xvals, yvals);
    end

    # Coordinates of points inside GO[3]
    for i = 2:rowNodes
        xvals = collect(-1:step1:0); # x values from -1 to 1 with step1
        yvals = (0 + step1*(i - 1))*ones(rowNodes,1); # rowNodes*2-1 cause there is 1 common
point, increase from -1 with step1 once every iteration

        start_idx = GO[3][end-i+1,1];
        end_idx = GO[3][end-i+1,end];
        coords[start_idx:end_idx, :] = hcat(xvals, yvals);
```

```
    end

    # ---- Set Variables & Iterate and Populate ----------------------------

    # Interpolation functions
    N = [
        (ksi, h) -> (1/4)*(1-ksi)*(1-h);
        (ksi, h) -> (1/4)*(1+ksi)*(1-h);
        (ksi, h) -> (1/4)*(1+ksi)*(1+h);
        (ksi, h) -> (1/4)*(1-ksi)*(1+h);
    ]

    # Assemble Stiffness Matrix & Right Hand Side
    Me = zeros(4, 4); # Element Me
    Te = zeros(4, 4); # Element Te

    ia = zeros(Int, 16*noe); # Row Sparse idx
    ja = zeros(Int, 16*noe); # Column Sparse idx
    va = zeros(Float64, 16*noe); # Value Sparse idx

    fe = zeros(4); # Element f
    F_global = zeros(nop, 1); # Global F

    c = 1; # idx for Sparse construction
    for e = 1:noe
        # Point Coordinates of each element
        xe = coords[Int.(l2g[e,:]), 1];
        ye = coords[Int.(l2g[e,:]), 2];

        # Jacobian matrix values quadrilateral elements
        J11 = (ksi, h) -> (1/4)*(-(1-h)*xe[1] + (1-h)*xe[2] + (1+h)*xe[3] - (1+h)*xe[4]);
        J12 = (ksi, h) -> (1/4)*(-(1-h)*ye[1] + (1-h)*ye[2] + (1+h)*ye[3] - (1+h)*ye[4]);
        J21 = (ksi, h) -> (1/4)*(-(1-ksi)*xe[1] - (1+ksi)*xe[2] + (1+ksi)*xe[3] + (1-
ksi)*xe[4]);
        J22 = (ksi, h) -> (1/4)*(-(1-ksi)*ye[1] - (1+ksi)*ye[2] + (1+ksi)*ye[3] + (1-
ksi)*ye[4]);
        detJ = (ksi, h) -> J11(ksi,h)*J22(ksi,h) - J12(ksi,h)*J21(ksi,h); # Determinant of J
matrix

        # Interpolation functions derivatives
        dNx = [
            (ksi, h) -> (1/(4*detJ(ksi, h))) * (-J22(ksi, h)*(1-h) + J12(ksi, h)*(1-ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * ( J22(ksi, h)*(1-h) + J12(ksi, h)*(1+ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * ( J22(ksi, h)*(1+h) - J12(ksi, h)*(1+ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * (-J22(ksi, h)*(1+h) - J12(ksi, h)*(1-ksi));
        ]

        dNy = [
            (ksi, h) -> (1/(4*detJ(ksi, h))) * ( J21(ksi, h)*(1-h) - J11(ksi, h)*(1-ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * (-J21(ksi, h)*(1-h) - J11(ksi, h)*(1+ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * (-J21(ksi, h)*(1+h) + J11(ksi, h)*(1+ksi));
            (ksi, h) -> (1/(4*detJ(ksi, h))) * ( J21(ksi, h)*(1+h) + J11(ksi, h)*(1-ksi));
        ]

        for i = 1:4 # 1:4 because we have 4 nodes per element
            for j = 1:4
                # Assemble Stiffness Matrix
```

```
                Mfunc = (ksi, h) -> (dNx[i](ksi, h)*dNx[j](ksi, h) + dNy[i](ksi, h)*dNy[j](ksi,
h)) * detJ(ksi, h);
                Me[i,j] = - DGQ_leg(Mfunc, 3);

                Tfunc = (ksi, h) -> k^2 * N[i](ksi, h) * N[j](ksi, h) * detJ(ksi, h);
                Te[i,j] = DGQ_leg(Tfunc, 3);
            end

            # Assemble right hand side
            xkh = (ksi, h) -> xe[1]*N[1](ksi, h) + xe[2]*N[2](ksi, h) + xe[3]*N[3](ksi, h) +
xe[4]*N[4](ksi, h);
            ykh = (ksi, h) -> ye[1]*N[1](ksi, h) + ye[2]*N[2](ksi, h) + ye[3]*N[3](ksi, h) +
ye[4]*N[4](ksi, h);
            ffunc = (ksi, h) -> N[i](ksi, h)*(f(xkh(ksi,h), ykh(ksi,h))) * detJ(ksi, h);
            fe[i] = DGQ_leg(ffunc ,3);

            # Global f
            F_global[Int.(l2g[e, i])] = F_global[Int.(l2g[e, i])] + fe[i];
        end

        # Global Sparse Striffness Matrix Construction
        for i = 1:4
            for j = 1:4
                ia[c] = l2g[e,i];
                ja[c] = l2g[e,j];
                va[c] = Me[i,j] + Te[i,j];
                c += 1;
            end
        end

    end

    K = sparse(ia, ja, va, nop, nop);

    # ---- Find & Enforce Dirichlet Boundary Conditions ---------------------------------------

    # Find the nodes
    boundary_nodes = findall(
        (coords[:,1] .== -1) .|
        (coords[:,1] .== 1) .|
        (coords[:,2] .== -1) .|
        (coords[:,2] .== 1) .|
        ((coords[:,1] .== 0) .& (coords[:,2] .>= 0)) .|
        ((coords[:,2] .== 0) .& (coords[:,1] .>= 0))
    );

    # Enforce the Boundary Conditions
    for i in boundary_nodes
        cols, _ = findnz(K[i, :])  # Ignore 2nd output of findnz (which is the actual non-zero
values themselves)
                                   # and keep only the columns indices of the non-zero elements
in row i
        for j in cols
            K[i, j] = 0.0
        end
        K[i, i] = 1.0
        F_global[i] = 0.0
```

```
    end

    return K, F_global, coords;
end
```

The function above takes as inputs the Global Ordering Matrix that was constructed with Lmesh.jl, our right-hand side function (-f, the "-" should be in the input or when first initializing the RHS (we initialized g = -f)) which was defined at Chapter 1 and the variable k, which represents the wave number, and is needed for our PDE. Then it computes the sparse Global Stiffness Matrix and the Global RHS based on the theory from above. The integrals are calculated from the double integral Legendre Gauss Quadrature function that you can find in *Appendix B*. Finally, it outputs the sparse Global Stiffness Matrix (K), the Global RHS (F_global), with the Dirichlet boundary conditions applied (refer to Chapter 1) and the coordinates of all the nodes inside the mesh (coords) (these coordinates were used to plot the domain at Chapter 2).

An example on how our main file should look like is presented below:

***Function 3**. Main.jl*

```
using Pkg
Pkg.activate(@__DIR__)
using MyFunctions, BenchmarkTools, SparseArrays, LinearAlgebra;
import PrettyTables;

step1 = 0.01;
GO = Lmesh(step1);

# Define the function as an anonymous function
s = 0.05;
k = 5.0; # If k = 0 then we have the Poisson Equation (Because we have division by
zero errors use eps(Float64) instead of 0). The bigger k is we have more peaks
f = (x, y) -> (-1/(2*pi*s^2)) * exp(-((x+0.5).^2 + (y-0.5).^2) / (2*s^2)) + (-
1/(2*pi*s^2)) * exp(-((x-0.5).^2 + (y+0.5).^2) / (2*s^2));

println("Building global FEM matrices (k=$k)...");
@time "Global FEM Assembly" K1, F_global1, all_coords = QuadFEM_Matrices(GO, f, k);
```

4. Sparse LU Solver for the Linear System

In this chapter we are going to build a solver for the linear system that we got as output from QuadFEM_Matrices.jl. The algorithm we are going to implement here is Sparse LU Decomposition algorithm and it goes as follows:

- Step 1:
  We have the system and the property:
  $$K \cdot x = b$$
  $$LU = R_s \cdot P \cdot K \cdot Q$$
  Where P and Q are the permutation matrices and Rs is the row scaling vector

- Step 2:
  We prepare the RHS:
  $$b' = R_s \cdot P \cdot b$$
  And we let:
  $$z = Q^{-1} \cdot x$$
  Then our system becomes:
  $$\begin{cases} K \cdot x = b \\ b = \dfrac{b'}{R_s \cdot P} \end{cases} \Rightarrow \begin{cases} K \cdot x = \dfrac{b'}{R_s \cdot P} \\ R_s \cdot P = \dfrac{LU}{K \cdot Q} \end{cases} \Rightarrow$$
  $$\Rightarrow LU \cdot Q^{-1} \cdot x = b' \Rightarrow \boldsymbol{LU \cdot z = b'}$$

- Step 3:
  We set in the last equation:
  $$U \cdot z = y$$
  And now we have to solve:
  $$L \cdot y = b'$$
  for “y” (Forward Substitution)

- Step 4:
  Now that we have “y” we solve:
  $$U \cdot z = y$$
  for “z” (Backward Substitution)

- Step 5:
  Finally recover “x”:
  $$x = Q \cdot z$$

The code that implements this is the following:

***Function 4**. LU_Solver.jl*

```
"""
    LU_Solver(K::SparseMatrixCSC{Float64, Int64}, F_global::Matrix{Float64})

This function solves a Sparse Linear System, K * x = F, using LU Factorization and
outputs
the solution

# Arguments
- **K::SparseMatrixCSC{Float64, Int64} :** The Sparse Global Stiffness Matrix of
our PDE
- **F_global::Matrix{Float64} :** The Global RHS of our PDE

# Returns
- **X_solution :** This is result after solving the system
"""
function LU_Solver(K::SparseMatrixCSC{Float64, Int64}, F_global::Matrix{Float64})
    F = lu(K);
    L = F.L; # Unit Lower Triangular Sparse Matrix
    U = F.U; # Upper Triangular Sparse Matrix
    p = F.p; # Row permutation vector
    q = F.q; # Column permutation vector
    Rs = F.Rs; # Row scaling vector

    n = size(K, 1);
    num_rhs = size(F_global, 2);
    X_solution = Matrix{Float64}(undef, n, num_rhs);

    # This single working vector will be transformed in-place:
    # F_col -> b' -> y_col -> z_col
    work_vector = Vector{Float64}(undef, n);

    for k_rhs = 1:num_rhs
        # Step 1) Prepare RHS: work_vector = b' = Rs * P * b
        for i = 1:n
            work_vector[i] = Rs[i] * F_global[p[i], k_rhs];
        end

        # Step 2) We have L * U * z = b' = work_vector
        # Let U * z = y -> Solve L * y = b' for y (Forward Substitution)
        for j = 1:n  # Iterate through columns of L
            val_y_j = work_vector[j]; # This y_j is final (as L_jj=1)
            if val_y_j != 0.0
                for k_idx = L.colptr[j]:(L.colptr[j+1]-1)
                    i_row = L.rowval[k_idx];
                    if i_row > j # Element L_ij is below the diagonal
                        work_vector[i_row] -= L.nzval[k_idx] * val_y_j;
                    end
                end
            end
        end
        # Now, work_vector = y

        # Step 3) U * z = y = work_vector -> Solve for z (Backward Substitution)
```

```
        for j = n:-1:1  # Iterate backwards through columns of U

            # Find U_jj (diagonal element of U in column j)
            U_jj = 0.0;

            for k_idx = U.colptr[j]:(U.colptr[j+1]-1)
                if U.rowval[k_idx] == j
                    U_jj = U.nzval[k_idx];
                    break
                end
            end

            current_sum_val = work_vector[j]; # This is y_j - sum(U_jk * z_k for
k>j)

            if U_jj == 0.0
                work_vector[j] = NaN; # Matrix is singular
            else
                work_vector[j] = current_sum_val / U_jj;
            end

            val_z_j = work_vector[j]; # This is the final value for z_j

            if val_z_j != 0.0 && !isnan(val_z_j)
                for k_idx = U.colptr[j]:(U.colptr[j+1]-1)
                    i_row = U.rowval[k_idx];
                    if i_row < j # Element U_ij is above the diagonal
                        work_vector[i_row] -= U.nzval[k_idx] * val_z_j;
                    end
                end
            end
        end
        # Now, work_vector = z

        # Step 4) x = Q * z -> Apply inverse column permutation -> Get solution x
        current_X_col = view(X_solution, :, k_rhs)
        for i = 1:n
            current_X_col[q[i]] = work_vector[i];
        end
    end

    return X_solution
end
```

This function takes as its 2 inputs the matrices of the linear system we want to solve, it computes the necessary L, U, P and Q components with the help of the function lu() (from the package LinearAlgebra.jl) and performs the algorithm we talked about in the beginning of this Chapter. Finally, it outputs the solution vector of the wanted system.

The results of out PDE, visualized, for different values of “k” are presented in the Figures below (for the plotting function check *Appendix C*):

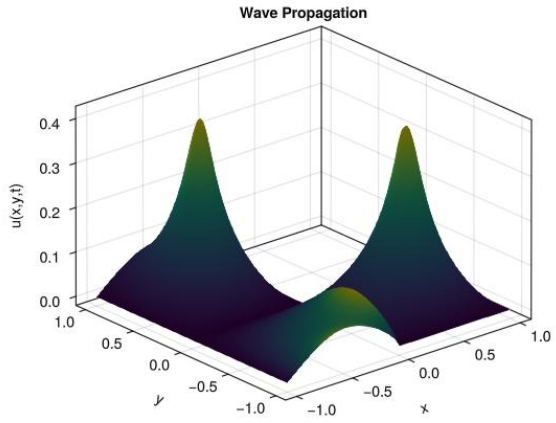


Figure 5. *Solution of the Helmholtz PDE for k = 0.01 (step=0.01), click here to see the animation*

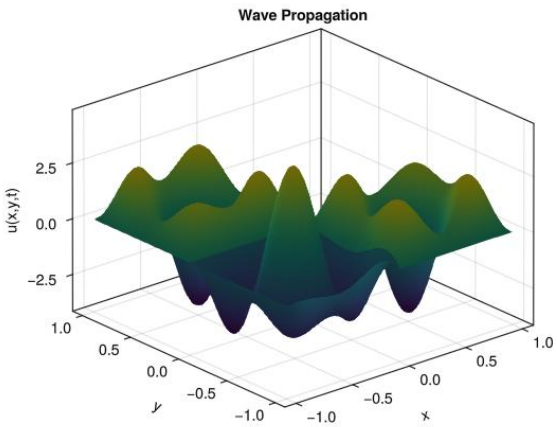


Figure 6. *Solution of the Helmholtz PDE for k = 10.0 (step=0.01), click here to see the animation*

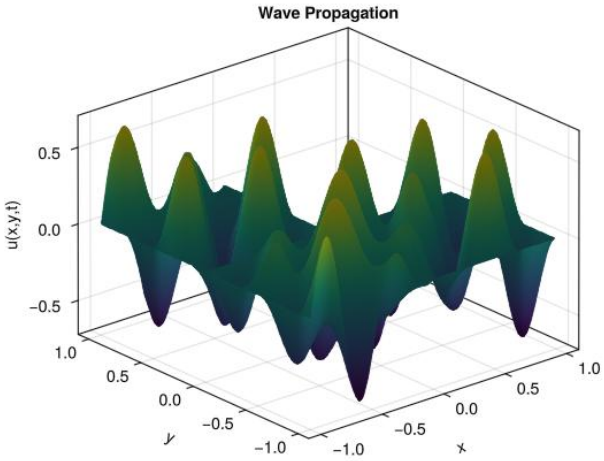


Figure 7. *Solution of the Helmholtz PDE for k = 15.0 (step=0.01), click here to see the animation*

The mean solving times of the solutions above can be viewed in the table presented below:

step = 0.01	LU_Solver (mean time)
k = 0.01	128.356 ms
k = 10.0	134.617 ms
k = 15.0	134.983 ms

5. Conjugate Gradient - GMRes Solver

In the pursuit of improving the performance of our solvers we turn to Krylov Subspace methods and specifically to the Conjugate Gradient method (or simply CG) and to the Generalized Minimal Residual method (or GMRes). The CG method, though, is guaranteed to converge only for SPD systems and in our case the K Stiffness matrix is neither Symmetric (the symmetry is ruined by the Dirichlet boundary conditions) nor Positive Definite so the obvious answer would be to go to GMRes. However, for small values of k (k = 0.01 or eps()), CG could still work would probably be a better choice, since CG is generally a more performance efficient method and since K would be close to an SPD. To decide which is better we are going to evaluate them both.

The function needed for the Conjugate Gradient is the following:

***Function 5**. Conj_Grad.jl*

```
"""
    Conj_Grad(A, B, tol)

Return the solution of 'A * x = B' using the Conjugate Gradient Method (CG)
"""
function Conj_Grad(A, B, tol, Nmax)
    M = size(A)
    q = zeros(M[1])

    # Residual
    r = B - A * q
    p = r

    # Norm of rbs
    norm0 = norm(r)

    if norm(r) > tol * norm0
        for i in 1:Nmax
            denom = dot(p, A*p);
            @assert abs(denom) > eps() "CG: Broke down in alpha division
(division by 0 = Inf)"
            a = dot(r, r) / denom; # dot(a, b) computes (a' * b)
            q += a * p;

            temp_r = r;
            r = r - a * A * p;

            if norm(r) < tol * norm0
                println("CG converged with ", i, " iterations");
                return q
            end
```

```
            denom2 = dot(temp_r, temp_r);
            @assert abs(denom2) > eps() "CG: Broke down in beta division
(division by 0 = Inf)"
            beta = dot(r, r) / denom2;
            p = r + beta * p;
        end
    end

    println("CG did not converge after $Nmax iterations. Final residual: ",
norm(r) / norm0)
    return q
end
```

This function takes as input the matrices of the linear system (let's say A and B), the tolerance and the maximum number of iterations it's allowed to perform. Then it computes the solution of the linear system A*x = B using the Conjugate Gradient algorithm.

For the GMRes we need these 2 functions:

***Function 6**. GMRes.jl*

```
"""
    GMRes(A::SparseMatrixCSC, b_matrix::Matrix{Float64}, tol::Float64;
max_restarts::Int = 300,restart_length::Int = min(30, size(A,1)))

Return the solution of 'A * x = B' using the Generalized Minimal Residual method
(GMRes)
"""
function GMRes(
    A::SparseMatrixCSC,
    b_matrix::Matrix{Float64},
    tol::Float64;
    max_restarts::Int = 300,
    restart_length::Int = min(30, size(A,1))
    )

    b = vec(b_matrix);
    n = size(A, 1);
    total_iterations = 0;

    x = zeros(Float64, size(A,1));

    # Initial residual
    r = b - A * x;
    norm0 = norm(r);

    if norm(r) < tol
        println("GMRES: Initial guess is already a solution (residual norm <
tol).");
```

```
        return 0
    end

    for restart_iter = 1:max_restarts
        rk = b - A * x;
        beta = norm(rk);

        V = zeros(Float64, n, restart_length + 1);
        H = zeros(Float64, restart_length + 1, restart_length);
        V[:, 1] = rk / beta;

        m = Arnoldi_ModGM(V, H, A, restart_length);
        total_iterations += m;

        Hm = H[1:(m+1), 1:m];
        e1_rhs = zeros(Float64, m + 1);
        e1_rhs[1] = 1.0;
        rhs_least_squares = beta .* e1_rhs;
        ym = Hm \ rhs_least_squares;

        x .+= V[:, 1:m] * ym;
        r = b - A * x;

        if norm(r) <= tol * norm0 || norm(r) < tol
            println("GMRES converged with $total_iterations iterations");
            break
        end
        if restart_iter == max_restarts
            println("GMRES reached max restarts ($max_restarts).");
        end
    end
    return x
end
```

Function 7. *Arnoldi_ModGM.jl*

```
"""
    Arnoldi_ModGM(V::AbstractMatrix{Float64}, H::AbstractMatrix{Float64},
A::SparseMatrixCSC, m_krylov::Int)

This function is used by GMRes to compute the Hessenberg matrix H, the
Orthonormal basis vectors and the effective Krylov subspace dimension m

# Arguments
- **V::AbstractMatrix{Float64} :** The Orthonormal basis from GMRes function
- **H::AbstractMatrix{Float64} :** The Hessenberg matrix
- **A::SparseMatrixCSC :** This would be our stiffness matrix K
- **m_krylov::Int :** Krylov subspace dimension m

# Returns
```

```
- **k_arnoldi :** The new Krylov subspace dimension m
"""
function Arnoldi_ModGM(
    V::AbstractMatrix{Float64},
    H::AbstractMatrix{Float64},
    A::SparseMatrixCSC,
    m_krylov::Int)

    k_arnoldi = m_krylov;

    for j = 1:m_krylov
        w = A * V[:, j];

        # Modified Gram-Schmidt
        for i = 1:j
            H[i, j] = dot(V[:, i], w);
            w .-= H[i, j] .* V[:, i];
        end

        H[j+1, j] = norm(w);

        if abs(H[j+1, j]) < 1e-12 # Breakdown tolerance
            k_arnoldi = j;
            break
        end
        V[:, j+1] = w / H[j+1, j];
    end
    return k_arnoldi
end
```

The GMRes.jl function takes, necessarily, as inputs the 2 matrices of the system (A and b_matrix) and the tolerance (tol) and then optionally you can add the maximum restarts that the algorithm is allowed to perform and the Krylov subspace dimensions. Arnoldi_ModGM.jl inputs are generated from GMRes.jl and are the Orthonormal basis, the Hessenberg matrix, the same sparse matrix that GMRes had for an input and finally the Krylov subspace dimension m. This, in particular, is the modified Gram-Schmit version of the Arnoldi iteration. These 2 functions work together to construct the GMRes algorithm.

Below you can see the tables with the mean solving times and the iterations of each function for different values of step and k:

Table 1

step = 0.5	CG mean time	CG iterations	GMRes mean time	GMRes iterations
k = 0.01	157.8 μs	3	115 μs	3
k = 1.0	160.8 μs	3	118.3 μs	3
k = 5.0	160.7 μs	3	138.75 μs	3

Table 2

step = 0.1	CG mean time	CG iterations	GMRes mean time	GMRes iterations
k = 0.01	240.5 μs	22	736.05 μs	30
k = 1.0	355.3 μs	22	757.1 μs	30
k = 5.0	435.2 μs	31	761.65 μs	30

Table 3

step = 0.01	CG mean time	CG iterations	GMRes mean time	GMRes iterations
k = 0.01	148.281 ms	217	841.547 ms	390
k = 1.0	163.577 ms	220	1.001 s	420
k = 5.0	205.518 ms	300	39.420 s	500

The red values mean that it reached the maximum iterations and didn't converge.

From a theoretical standpoint, since our system is not SPD CG lacks theoretical guarantees for convergence to the correct solution and can lead to breakdown or erratic behavior. GMRes on the other hand, is designed for non-symmetric and indefinite matrices and is the theoretically correct choice for our problem. However, after analyzing the data given in the tables above, we can see that for a very coarse mesh (Table 1) both functions converge very fast and with similar mean times (GMRES exhibits slightly faster mean times) but as we move to a more medium and fine mesh GMRes seems to struggle in comparison with CG. This can probably be attributed to GMRes's expensive orthogonalization steps (Arnoldi) which take longer times per iteration. Furthermore, when we reach the finest mesh (Table 3) we observe that GMRes for k = 5.0 doesn't converge at all (when it is theoretically the correct choice) after 500 iterations (it was stopped at 500 iterations) and CG has generally a lot higher iteration counts. This underscores the critical need for effective preconditioning. For the preconditioning of these functions we will talk about in one of the next Chapters.

6. Domain Decomposition with 3 Subdomains and each Stiffness Matrix – RHS

In this Chapter we are going to make a function that takes one of the outputs of the QuadFEM_Matrices.jl, the GO matrix which contains the Global Ordering matrix of each of the 3 subdomains, in order to make the subdomains overlap with each other by a specific number of elements. Specifically, it is going to be able to make each subdomain overlap its neighboring subdomains based on a value H which will represent the overlap in elements. Furthermore, it will be able to output Stiffness matrix and the RHS vector that corresponds to each subdomain. The function which will be able to do all that is presented below:

Function 8. *SubdomainOverlap_Matrices.jl*

```
"""
    SubdomainOverlap_Matrices(GO::Vector{Matrix{Int64}}, K::SparseMatrixCSC{Float64,
Int64} , F_global::Matrix{Float64}, H::Int)

This function takes the Global Ordering, in subdomains, and overlaps each into the
other by H elements. Then
it constructs and returns the stiffness and right hand side matrices of each subdomain.

# Arguments
- **GO::Vector{Matrix{Int64}} :** The Global Ordering matrix which contains the 3
seperate subdomains matrices meshed (use Lmesh to create it)
- **K::SparseMatrixCSC{Float64, Int64} :** K is the sparse stiffness matrix of our
inhomogeneous Helmholtz equation
- **F_global::Matrix{Float64} :** F_global is the final right hand side of the
equation. Now use a solver to solve the system K*u=F_global for u(x,y,t)
- **H::Int :** This variable sets the overlap number, in elements, between the
subdomains

# Returns
- **K_sub :** This contains the stiffness matrices for each subdomain (outputs:
K_sub[i], i=1,2,3)
- **F_global_sub :** This contains the right hand side matrices for each subdomain
(outputs: F_global_sub[i], i=1,2,3)
- **idx :** The 3 subdomains node vectors with their overlaps
"""
function SubdomainOverlap_Matrices(GO::Vector{Matrix{Int64}},
                                   K::SparseMatrixCSC{Float64, Int64},
                                   F_global::Matrix{Float64},
                                   H::Int)

    # Catch Error: Overlap must not exceed the element length of a subdomains vertical
or horizontal side
    if H > length(GO[1][:,1]) - 1
        return error("Overlap H must NOT exceed the element length of a subdomains
vertical or horizontal side");
    end
```

```
    # Inner node vectors
    inGO1 = vec(GO[1]);
    inGO2 = vec(GO[2]);
    inGO3 = vec(GO[3]);

    # Overlapping node vectors
    G12 = vec(GO[2][:, 1:H+1]); # Overlap of subdomain 1 into 2 (H+1 because H is the
overlap in elements)
    G13 = vec(GO[3][end-H:end, :]); # Overlap of subdomain 1 into 3
    G1 = vcat(G12, G13); # Merge the overlapping nodes of subdomain 1 into 2 & 3
    G21 = vec(GO[1][:, end-H:end]); # Overlap of subdomain 2 into 1
    G31 = vec(GO[1][1:H+1, :]); # Overlap of subdomain 3 into 1

    # Overlapping and inner node union vectors
    idx1 = unique([inGO1;G1]);
    idx2 = unique([inGO2;G21]);
    idx3 = unique([inGO3;G31]);

    # Subdomain stiffness matrices
    K_sub = [
        K[idx1, idx1],
        K[idx2, idx2],
        K[idx3, idx3]
    ]

    # Subdomain right hand side
    F_global_sub = [
        F_global[idx1],
        F_global[idx2],
        F_global[idx3]
    ]

    return K_sub, F_global_sub, [idx1, idx2, idx3]
end
```

As we said in the prologue of this chapter, this function takes as input the GO vector which contains the 3 subdomain GO matrices, the global stiffness matrix (K), the global RHS (F_global) and the value H (which represents the overlap in elements). Then by computing the overlapping node vectors G it is able to extract from the global K and RHS the Stiffness matrix K_{sub} and the rhs $F_{global_{sub}}$ of each subdomain. Simultaneously, it also returns all the indexes of the overlapping and original nodes of each subdomain.

The main file of our project, now, would look something like this:

Function 9. *Main.jl*

```
using Pkg
Pkg.activate(@__DIR__)
using MyFunctions, BenchmarkTools, SparseArrays, LinearAlgebra;
import PrettyTables;
```

```
step1 = 0.01;
GO = Lmesh(step1);

# Define the function as an anonymous function
s = 0.05;
k = 5.0; # If k = 0 then we have the Poisson Equation (Because we have
division by zero errors use eps(Float64) instead of 0). The bigger k is we
have more peaks
f = (x, y) -> (-1/(2*pi*s^2)) * exp(-((x+0.5).^2 + (y-0.5).^2) / (2*s^2)) +
(-1/(2*pi*s^2)) * exp(-((x-0.5).^2 + (y+0.5).^2) / (2*s^2));

println("Building global FEM matrices (k=$k)...");
@time "Global FEM Assembly" K1, F_global1, all_coords = QuadFEM_Matrices(GO,
f, k);

H = 2; # Overlap
println("\nPerforming domain decomposition with overlap H = $H...")
@time "Subdomain Overlap" K_sub, F_sub, idx = SubdomainOverlap_Matrices(GO,
K1, F_global1, H);
println("Number of subdomains created: ", length(K_sub));

# --- Solving the System ---
tol = 10^-4;
Nmax = 500; # Max iterations
krylov_dim_gmres = 30;

println("\n--- Solving with k=$k ---")

@time "Solving time" begin
    # u = K1\F_global1;
    # u = LU_Solver(K1, F_global1);
    # u = Conj_Grad(K1, F_global1, tol, 1000);
    u = GMRes(K1, F_global1, tol, max_restarts = Nmax);
end


# ------------------------------------------------------------------------
#       Plotting the result
# ------------------------------------------------------------------------

PlotDomain(all_coords)
PlotDomainOverlaps(all_coords, idx)
PlotSolutionAnimation(GO, u, k)

nothing
```

The visualization of the overlapping can be achieved with the help of the function PlotDomainOverlaps() (refer to *Appendix D*) and it will give us these plots:

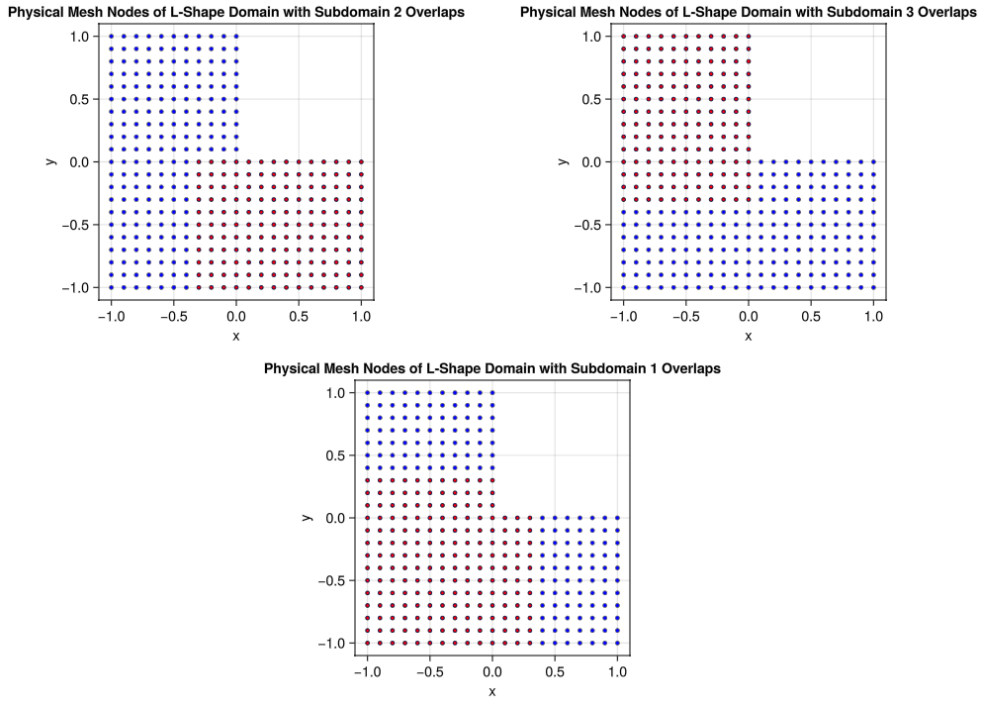


Figure 8. *Physical Mesh Nodes of L-Shape Domain with Subdomain Overlaps (step=0.1, H=3)*

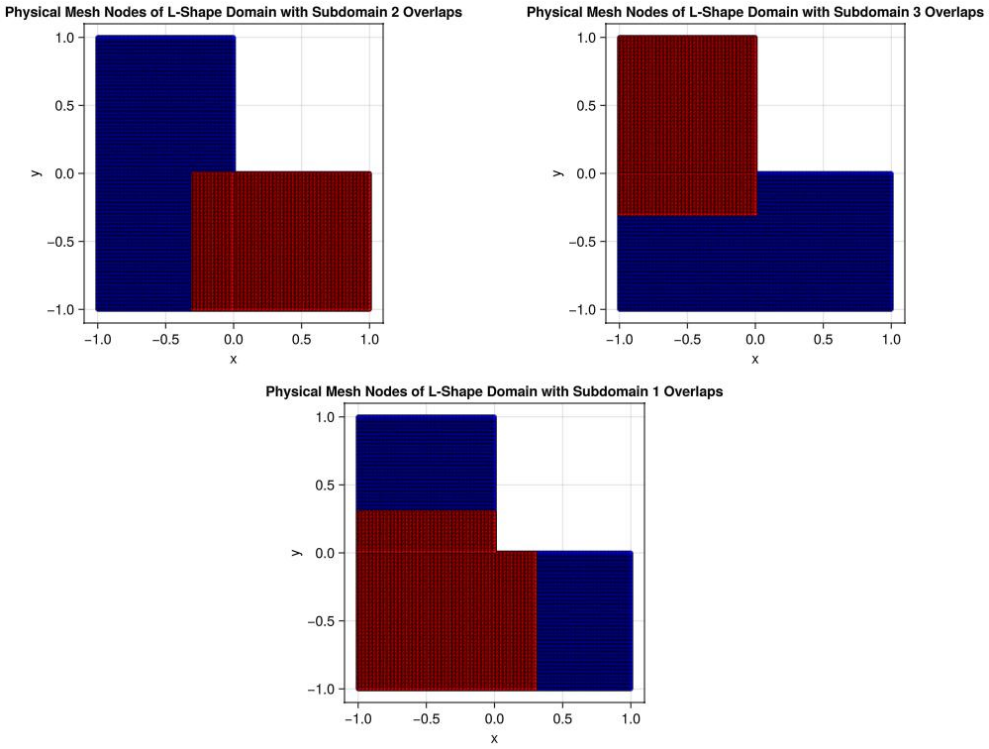


Figure 9. *Physical Mesh Nodes of L-Shape Domain with Subdomain Overlaps (step=0.01, H=30)*

7. Preconditioned CG – GMRes with SAP

As we saw in Chapter 5 from the Tables1,2,3, as we move to finer meshes, there is a critical need for preconditioning. Exactly that is what we are going to do in this Chapter. As a preconditioner to our functions, CG and GMRes, we are going to use a Domain Decomposition algorithm called Schwartz Additive Procedure (or SAP).

SAP takes the problem divided into subdomains (for our case 3 subdomains) and gives an approximate solution to a local problem of the form:

$$M_i \cdot z_i = r_i \Rightarrow z_i = M_i^{-1} \cdot r_i, \qquad i = 1, \dots, N_{subdomains}$$

Then after computing z_i for every subdomain it adds up all the solutions to one vector for the preconditioned (or pcd) function to use. The code for the SAP that will be used by both CG and GMRes as a preconditioner is presented below:

Function 10. *pcdSAP.jl*

```
"""
    pcdSAP(K::SparseMatrixCSC{Float64, Int64},
           r_input::Matrix{Float64},
           K_sub:: Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
           idx::Vector{Vector{Int}})

This function is a preconditioner, utilizing Schwartz Additive Procedure, that is
used
together with the preconditioned Conjugate Gradient Method (pcdCG function) and
GMRes (pcdGMRes function)

# Arguments
- **K::SparseMatrixCSC{Float64, Int64} :** K is the sparse stiffness matrix of
our inhomogeneous Helmholtz equation
- **r_input::Matrix{Float64} :** This is the residual input vector of the pcdCG
function
- **K_sub:: Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}} :** This
contains the stiffness matrices for each subdomain factorized (outputs: K_sub[i],
i=1,2,3)
- **idx::Vector{Vector{Int}} :** The 3 subdomains node vectors with their
overlaps

# Returns
 - **z :** This is the solution of the system z = M^-1 * r , where M^-1 is the
preconditioner matrix
"""
function pcdSAP(K::SparseMatrixCSC{Float64, Int64},
                r_input::Matrix{Float64},
                K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
                idx::Vector{Vector{Int}}
                )
```

```
    nop = size(K,1); # Number of nodes in the whole domain
    z = zeros(nop, 1); # Output vector initialization

    # Solve system for the union of subdomain 1 and its overlaps
    z1 = K_sub[1] \ r_input[idx[1], 1];
    # Solve system for the union of subdomain 2 and its overlaps
    z2 = K_sub[2] \ r_input[idx[2], 1];
    # Solve system for the union of subdomain 3 and its overlaps
    z3 = K_sub[3] \ r_input[idx[3], 1];

    z[idx[1], 1] .+= z1;
    z[idx[2], 1] .+= z2;
    z[idx[3], 1] .+= z3;

    return z;
end
```

As we can see it takes as inputs the global sparse Stiffness matrix K, the residual input vector r_{input}, the subdomains Stiffness matrix K_{sub} (for optimization they are received pre-factorized from the function factorize() of the LinearAlgebra.jl pkg to improve performance) and the index vector of the nodes of each subdomain (with the overlaps). Afterwards it solves all the subproblems z_i and adds them all to the common vector z. Finally, it returns the common vector z for the preconditioned function to use.

The modified CG that supports the preconditioner is this:

Function 11. *pcdCG.jl*

```
"""
    pcdCG(K::SparseMatrixCSC{Float64, Int64},
              F::Matrix{Float64},
              tol::Float64,
              K_sub:: Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
              idx::Vector{Vector{Int}}, Nmax::Int)

This function is an implementation of a preconditioned CG Method (Polak-Ribière variant of
CG), using SAP (pcdSAP function) as a preconditioner

# Arguments
- **K::SparseMatrixCSC{Float64, Int64} :** K is the sparse stiffness matrix of our
inhomogeneous Helmholtz equation
- **F::Matrix{Float64} :** F is the final right hand side of the equation
- **tol::Float64 :** This is the tolerance of the solver (ex. 10^-6)
- **K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}} :** This contains the
stiffness matrices for each subdomain factorized
- **idx::Vector{Vector{Int}} :** The 3 subdomains node vectors with their overlaps
- **Nmax::Int :** The maximum number of iterations

# Returns
```

```
 - **q :** The solution of the system K * q = F
"""
function pcdCG(K::SparseMatrixCSC{Float64, Int64},
               F::Matrix{Float64},
               tol::Float64,
               K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
               idx::Vector{Vector{Int}}, Nmax::Int
               )

    M = size(K, 1);
    q = zeros(M[1]);

    # Initial Residual
    r = F - K * q;
    norm0 = norm(r); # Norm of initial r (r0)
    # Initial preconditioned residual (z0 = M^-1 * r0)
    z = pcdSAP(K, r, K_sub, idx);
    p = z;

    for i in 1:Nmax
        w = K * p;

        a_denom = dot(p, w);
        @assert abs(a_denom) > eps() "pcdCG: Broke down in alpha division (division by 0 =
Inf)"
        a = dot(r, z) / a_denom;
        q = q + a*p;
        r_new = r - a*w;

        current_norm = norm(r_new) / norm0;
        println("Iteration $i: ||r|| / ||r0|| = $current_norm");

        if current_norm < tol
            println("pcdCG converged with ", i, " iterations");
            return q
        end

        # New preconditioned residual ( z_k+1 = M^-1 * r_k+1)
        z_new = pcdSAP(K, r_new, K_sub, idx);

        denom2 = dot(r, z);
        @assert abs(denom2) > eps() "pcdCG: Broke down in beta division (division by 0 =
Inf)"
        beta = dot(r_new, z_new) / denom2;
        p = z_new + beta*p;

        # Update r and z
        r = r_new;
        z = z_new;
    end
    println("pcdCG did not converge after $Nmax iterations. Final residual: ", norm(r) /
norm0)
    return q
end
```

This function is an implementation of the Polak-Ribière variant of CG. The inputs of this function are identical to the regular CG function, Conj_Grad(), with only exception being that it also needs the subdomain Stiffness matrices as an input in order to pass it to pcdSAP().

The modified GMRes that supports the preconditioner is this:

Function 12. *pcdGMRes.jl*

```
"""
    pcdGMRes(A::SparseMatrixCSC, F_matrix::AbstractMatrix{Float64}, tol::Float64,
            Nmax_restarts::Int, K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64,
Int64}},
            idx::Vector{<:Vector{<:Integer}}; restart_len::Int = min(30, size(A,1)),
            x0_vec::AbstractVector{Float64} = zeros(Float64, size(A,1))
            )

This function an implementation of a preconditioned GMRes method, using SAP (pcdSAP
function) as a preconditioner

# Arguments
- **A::SparseMatrixCSC :** The sparse stiffness matrix of our PDE
- **F_matrix::AbstractMatrix{Float64} :** The RHS of our system
- **tol::Float64 :** This is the tolerance of the solver (ex. 10^-6)
- **Nmax_restarts::Int :** This is the number of maximum allowed iterations
- **K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}} :** This contains the
stiffness matrices for each subdomain factorized
- **idx::Vector{<:Vector{<:Integer}} :** The 3 subdomains node vectors with their overlaps
- **restart_len::Int = min(30, size(A,1)) :** Krylov subspace dimension m
- **x0_vec::AbstractVector{Float64} = zeros(Float64, size(A,1)) :** Initial guess

# Returns
- **x :** The solution of the linear system K * x = F
"""
function pcdGMRes(
    A::SparseMatrixCSC,
    F_matrix::AbstractMatrix{Float64},
    tol::Float64,
    Nmax_restarts::Int,
    K_sub::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
    idx::Vector{<:Vector{<:Integer}};
    restart_len::Int = min(30, size(A,1)), # Krylov subspace dimension m
    x0_vec::AbstractVector{Float64} = zeros(Float64, size(A,1)) # Initial guess
    )

    n = size(A, 1);
    x = copy(x0_vec); # Initial guess vector
    b_vec = vec(F_matrix); # Ensure RHS is a 1D vector

    total_arnoldi_steps = 0;

    initial_r = b_vec - A * x;
    norm0 = norm(initial_r);

    println("Starting GMRES_with_pcdSAP: Max Restarts = $Nmax_restarts, Restart Length =
$restart_len, Tolerance = $tol");
```

```
    println("Restart 0 (Initial state): RelRes (to initial) approx 1.0, AbsRes = $norm0");

    # Define the preconditioner application function using the wrapper function
    apply_preconditioner = r_v -> pcdSAP_vector_interface_wrapper(A, r_v, K_sub, idx);
    # Define the effective operator for Arnoldi: M^-1 * A * v
    effective_A_operator = v_for_mv -> apply_preconditioner(A * v_for_mv);

    for restart_iter = 1:Nmax_restarts
        # Current unpreconditioned residual r_k = b - A*x_k
        upcd_rk = b_vec - A * x;
        # Starting vector for Arnoldi: r_k = M^-1 * upcd_rk
        rk = apply_preconditioner(upcd_rk);
        beta = norm(rk);

        if abs(beta) < 1e-14 # Break if preconditioned residual is (near) zero
            println("GMRES_with_pcdSAP: Preconditioned residual norm ($beta) is effectively
zero at restart $restart_iter.");
            break
        end

        Vm = zeros(Float64, n, restart_len + 1); # Initialize Vm
        H = zeros(Float64, restart_len + 1, restart_len); # Hessenberg
        Vm[:, 1] = rk / beta;

        m_eff = pcdArnoldi(Vm, H, effective_A_operator, restart_len);
        total_arnoldi_steps += m_eff;

        if m_eff == 0; break; end

        Hm = H[1:(m_eff+1), 1:m_eff];
        e1 = zeros(Float64, m_eff + 1); e1[1] = 1.0;
        rhs_ls = beta .* e1;
        ym = Hm \ rhs_ls; # Least squares solve for ym (length m_eff)

        x .+= Vm[:, 1:m_eff] * ym; # Update solution: x_new = x_old + V_m * y_m

        # Convergence check -> Relative residual to initial unpreconditioned residual
        r = b_vec - A * x;
        relative_res = norm(r) / norm0;

        println("GMRES_with_pcdSAP Restart #$restart_iter (Arnoldi dim $m_eff):
RelResToInitial = $relative_res, AbsRes = $(norm(r))");

        if norm(r) < tol || relative_res < tol
            println("GMRES_with_pcdSAP converged with $total_arnoldi_steps iterations");
            break
        end
        if restart_iter == Nmax_restarts
            println("GMRES_with_pcdSAP reached max restarts ($Nmax_restarts).");
        end
    end

    return x
end
```

This pcd version of GMRes has all the inputs of the normal GMRes and 3 extra ones. The first 2, extra ones, are the subdomain stiffness matrix K_{sub} and the subdomain node vectors idx, that are going to be needed for the pcdSAP() inside pcdGMRes(). The final input is the initial guess $x0_{vec}$ which defaults to 0 if we don't set it to anything. The main structure of the algorithm is the same as the normal GMRes with only difference being that it calls the preconditioned Arnoldi iteration, pcdArnoldi(), instead of the normal one. The code for the pcdArnoldi() is presented below:

Function 13. *pcdArnoldi.jl*

```
"""
    pcdArnoldi(V::AbstractMatrix{Float64}, H::AbstractMatrix{Float64},
                effective_operator_apply::Function, m_krylov::Int)

# Arguments
- **V::AbstractMatrix{Float64} :** Orthonormal basis (n x (m_krylov+1))
- **H::AbstractMatrix{Float64} :** Hessenberg matrix (m_krylov+1 x m_krylov)
- **effective_operator_apply::Function :** Function to apply the effective operator
- **m_krylov::Int :** Desired Krylov subspace dimension

# Returns
- **k_actual :** Actual achieved dimension of the Krylov subspace constructed during that
particular Arnoldi process
"""
function pcdArnoldi(
    V::AbstractMatrix{Float64},
    H::AbstractMatrix{Float64},
    effective_operator_apply::Function,
    m_krylov::Int
    )
    k_actual = m_krylov;

    for j = 1:m_krylov
        w = effective_operator_apply(V[:, j]); # w = EffectiveOp * v_j
        # Modified Gram-Schmidt
        for i = 1:j
            H[i, j] = dot(V[:, i], w);
            w .-= H[i, j] .* V[:, i];
        end

        H[j+1, j] = norm(w);

        if abs(H[j+1, j]) < 1e-12 # Breakdown tolerance
            k_actual = j;
            break
        end
        # Only fill V[:,j+1] if we are not at the last iteration of this m_krylov cycle
        if j <= m_krylov # Check to prevent writing V[:,m_krylov+2] if m_krylov was used in
V size
             V[:, j+1] = w / H[j+1, j];
        end
    end
    return k_actual
end
```

As we can see the pcdArnoldi() has effectively the same algorithmic structure as Arnoldi_ModGM() (with only difference being the way we calculate w) and has almost all the same inputs as Arnoldi_ModGM() did. The only difference is that here instead of A (the Stiffness matrix) the new Arnoldi inputs a function that will help us to make pcdGMRes() and pcdArnoldi() compatible with the already existing pcdSAP(). This new function is called a "wrapper" function and is shown below:

Function 14. *pcdSAP_vector_interface_wrapper.jl*

```
function pcdSAP_vector_interface_wrapper(
    K_global_for_pcdSAP::SparseMatrixCSC{Float64, Int64},
    r_input_vec::AbstractVector{Float64},
    K_sub_all::Vector{SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}},
    idx_all_subdomains::Vector{<:Vector{<:Integer}}
    )

    # Convert input 1D vector to an Nx1 matrix for the pcdSAP
    r_input_matrix = reshape(r_input_vec, :, 1);

    # Call pcdSAP function
    z_output_matrix = pcdSAP(K_global_for_pcdSAP, r_input_matrix, K_sub_all,
idx_all_subdomains);

    # Convert output Nx1 matrix back to a 1D vector
    return vec(z_output_matrix)
end
```

With this new function GMRes is now able to pass the computing of the residual $r_k = M^{-1} \cdot r_{k_{unpcd}}$ to the preconditioner pcdSAP() and also make the preconditioner accessible from inside of the pcdArnoldi() functions where it's needed to calculate $w = M^{-1} \cdot A \cdot V_j$.

Now with our functions ready we can proceed to collect some data. We will benchmark and compare the preconditioned, with SAP, functions CG and GMRes against each other and against backslash (u=K\F, the default solver of Julia) for different values of the wavenumber k, the step and the overlap H in elements. For the benchmarking we will use the BenchmarkTools.jl pkg for Julia.

Table 4

step = 0.1	pcdCG		pcdGMRes		K\F
	Mean Time	Iterations	Mean Time	Iterations (Arnoldi)	Mean Time
k=5.0, H=1	1.138 ms	7	1.723 ms	30	
k=5.0, H=2	1.095 ms	7	1.690 ms	30	597.900 μs
k=5.0, H=5	1.233 ms	7	1.823 ms	30	
k=15.0, H=1	1.786 ms	12	1.665 ms	30	
k=15.0, H=2	1.906 ms	12	1.692 ms	30	605.700 μs
k=15.0, H=5	2.024 ms	13	1.932 ms	30	
k=35.0, H=1	3.082 ms	21	1.662 ms	30	
k=35.0, H=2	3.424 ms	23	1.726 ms	30	623.000 μs
k=35.0, H=5	3.882 ms	26	1.889 ms	30	

Table 5

step = 0.01	pcdCG		pcdGMRes		K\F
	Mean Time	Iterations	Mean Time	Iterations (Arnoldi)	Mean Time
k=5.0, H=10	46.791 ms	9	190.674 ms	30	
k=5.0, H=20	47.281 ms	8	213.957 ms	30	94.266 ms
k=5.0, H=40	61.632 ms	8	242.126 ms	30	
k=15.0, H=10	76.321 ms	15	185.609 ms	30	
k=15.0, H=20	90.431 ms	14	206.496 ms	30	94.788 ms
k=15.0, H=40	118.838 ms	16	248.020 ms	30	
k=35.0, H=10	211.421 ms	37	193.499 ms	30	
k=35.0, H=20	371.059 ms	59	635.072 ms	90	96.601 ms
k=35.0, H=40	344.533 ms	43	1.595 s	180	

From these results (Table 4 & 5) we observe that pcdCG (PCG preconditioned with SAP) offers excellent performance for a range of wavenumbers (k) and mesh sizes, likely due to the strength of the SAP preconditioner mitigating the theoretical issues of applying PCG to a non-SPD system. Nonetheless, for higher k values, in a medium sized mesh (Table 4, step=0.1, k>5.0) pcdGMRes shows its robustness and outperforms pcdCG. This, however, doesn't stay true for long since when we move the finer meshes (Table 5, step=0.01) the higher cost per iteration of GMRes becomes evident and drags the method back.

Furthermore, we also have to note the impact of the overlap, H, in the performance of both methods. Generally, a larger overlap provides more information to each subdomain system, which can lead to a more effective preconditioner and faster convergence in terms of iterations. However, as we can see from Tables 4 & 5 if H is set to a very large value, compared to the subdomain mesh size, this can hurt our performance instead of improving

it. This is more notable at Table 5 as pcdCG, for k = 15, goes from H = 10 to H = 40 where iterations decrease (from 15 to 14 and then 16) but the solving time increases (from 76ms to 90ms and then 118ms). This illustrates that the computational cost of the larger subdomain systems can outweigh the benefit of fewer iterations. Therefore, we must find the optimal value of H in order to balance out the iterations and solving time and achieve the best performance out of our methods.

Ultimately, both methods demonstrate the effectiveness of the SAP preconditioner (pcdSAP). The choice between pcdCG and pcdGMRes as the outer Krylov solver can depend on the specific target k value, with pcdCG offering speed for many cases and pcdGMRes providing a robustness advantage for the most challenging ones.

8. Multigrid as a Preconditioner in CG

Since, generally, in the benchmark results of Chapter 7 (Table 4 & 5) CG outperformed GMRes and converged we are going to implement the Multigrid method in the SAP preconditioner of CG in order to try to improve its performance.

The first thing we are going to construct is the function that handles the Multigrid V-cycle.

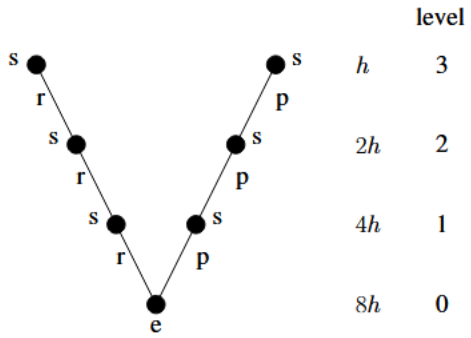


Figure 10. *Multigrid V-cycle*

The functions that will handle that are:

Function 15. *MG_Vcycle.jl*

```
function MG_Vcycle(
    b::Vector{Float64},
    mg::MGHier,
    nPre::Int,
    nPost::Int,
    omega::Float64,
    x0_guess::Vector{Float64}=zeros(Float64, length(b)),
    lvl_idx::Int = 1,
    )

    lev_data = mg.levels[lvl_idx];
    A = lev_data.A;

    # Coarsest level check
    if lvl_idx == length(mg.levels) || lev_data.R == nothing
        if size(A,1) > 0
            return A \ b
        else # A is empty, b should also be empty
            return zeros(Float64,0)
        end
    end

    # Initialize Solution
    x = copy(x0_guess);
```

```
    # Pre-smoothing (Damped Jacobi)
    Dinv = 1.0 ./ diag(A);
    for _ = 1:nPre
        res_smooth_pre  = b - A * x;
        x .+= omega .* (res_smooth_pre .* Dinv);
    end

    # Restrict
    rFine = b - A * x;
    rCoarse = lev_data.R * rFine;

    # Recursive correction
    e_coarse_initial = zeros(Float64, length(rCoarse));
    e_coarse = MG_Vcycle(rCoarse, mg, nPre, nPost, omega, e_coarse_initial,
                         lvl_idx + 1);

    # Prolongate and correct
    if !isempty(e_coarse) # Check if coarse error is not empty
        x .+= lev_data.P * e_coarse
    end

    # Post-smoothing
    for _ = 1:nPost
        res_smooth_post = b - A * x;
        x .+= omega .* (res_smooth_post .* Dinv)
    end
    return x
end
```

Function 16. *MGLevel.jl*

```
struct MGLevel
    A::SparseMatrixCSC{Float64, Int64} # Stiffness matrix
    R::Union{SparseMatrixCSC{Float64, Int64}, Nothing} # Restriction (nothing for
coarsest)
    P::Union{SparseMatrixCSC{Float64, Int64}, Nothing} # Prolongation (nothing
for coarsest)
    nodes_original_indices::Union{Vector{Int}, Nothing}
    coords::Union{Matrix{Float64}, Nothing}
end

struct MGHier
    levels::Vector{MGLevel}
end
```

Function 15. (MG_Vcycle.jl) has for inputs the RHS of each subdomain (b), an instance of the Multigrid Hierarchy that is stored with the help of our struct MGHier (mg), the number of pre-smoothing and post-smoothing iterations of the inlined Damped Jacobi (nPre, nPost), the dampening factor of Damped Jacobi (omega), an initial guess to the problem it solves (x0_guess, it defaults at 0) and finally the level that is currently at in the Multigrid cycle (lvl_idx). With those inputs it performs a standard V cycle of the Multigrid algorithm in order to solve the system it was given to it.

The MG_Vcycle.jl function will now need to be called by a new modified version of the preconditioned SAP function which will lie inside a new modified preconditioned CG function.

Function 17. *MGpcdSAP.jl*

```
function MGpcdSAP(
    K_global::SparseMatrixCSC{Float64, Int64},
    r_input::AbstractVector{Float64},
    idx::Vector{Vector{Int}},
    mg_hier::Vector{MGHier},
    nPre::Int,
    nPost::Int,
    omega::Float64
    )

    nop = size(K_global,1);
    z = zeros(Float64, nop);

    for i = 1:3
        # Get subdomain rhs
        b_sub = r_input[idx[i]];

        z_sub = MG_Vcycle(b_sub, mg_hier[i], nPre, nPost, omega);
        z[idx[i]] .+= z_sub;
    end

    return z
end
```

Function 18. *MGpcdCG.jl*

```
function MGpcdCG(
    K::SparseMatrixCSC,
    F::AbstractMatrix{Float64},
    tol::Float64,
    idx::Vector{Vector{Int64}},
    mg_hier::Vector{MGHier},
    Nmax::Int,
    omega::Float64 = 4/5, # Default smoother omega
```

```
    nPre::Int = 2, # Default number of pre-smoothing iterations
    nPost::Int = 1 # Default number of post-smoothing iterations
    )
    F_vec = vec(F);

    nop = size(K, 1);
    q = zeros(Float64, nop);

    r = F_vec - K * q; # Initial residual (vector)
    norm0 = norm(r);
    if norm0 < 1e-14
        println("MGpcdCG: Initial residual is zero. Solution is the initial guess.")
        return q
    end

    # Initial preconditioned residual (z0 = M^-1 * r0)
    z = MGpcdSAP(K, r, idx, mg_hier, nPre, nPost, omega);
    p = copy(z);

    rz_old = dot(r, z) # For Polak-Ribière beta

    println("Starting MGpcdCG: Max Iterations = $Nmax, Tolerance = $tol")
    println("Iteration 0: ||r||/||r0|| = 1.0")

    for iter in 1:Nmax
        w = K * p;

        a_denom = dot(p, w);
        @assert abs(a_denom) > eps() "MGpcdCG: Broke down in alpha division (division by 0
= Inf)"
        a = rz_old / a_denom;
        q .+= a .* p;
        r .-= a .* w; # r is now r_{k+1}

        current_rel_res = norm(r) / norm0
        println("Iteration $iter: ||r|| / ||r0|| = $current_rel_res")

        if current_rel_res < tol
            println("MGpcdCG converged in $iter iterations.");
            return q
        end

        # New preconditioned residual (z_{k+1} = M^-1 * r_{k+1})
        z_new = MGpcdSAP(K, r, idx, mg_hier, nPre, nPost, omega);

        @assert abs(rz_old) > eps() "MGpcdCG: Broke down in beta division (division by 0 =
Inf)"
        rz_new = dot(r, z_new);
        beta = rz_new / rz_old; # Polak-Ribière: (r_{k+1}^T z_{k+1}) / (r_k^T z_k)

        p = z_new .+ beta .* p;

        rz_old = rz_new;
    end

    println("MGpcdCG did not converge after $Nmax iterations. Final relative residual: ",
norm(r) / norm0)
```

```
    return q
end
```

As we can see the new versions of the preconditioned SAP and MG functions are mostly the same in principle as their older versions. The key differences derive from their extra inputs and those are used.

Starting from the outer layer of the solver, Function 18. (MGpcdCG.jl), takes as input the sparse Stiffness matrix of our system (K), the global RHS (F), a tolerance (tol) below which the method converges, the 3 subdomains node vectors with their overlaps (idx), a list of the Multigird hierarchies (mg_hier), the maximum number of iterations acceptable (Nmax), the dampening factor of Damped Jacobi (omega) and finally the number of pre and post smoothing iterations (nPre, nPost respectively). Once those are provided it functions exactly the same as Function 11. (pcdCG.jl) with the only difference being that it now calls the new, Multigrid oriented, version of Function 10. (pcdSAP.jl), Function 17. (MGpcdSAP.jl), to compute the preconditioned residual.

Moving deeper now, Function 17. (MGpcdSAP.jl), needs as arguments the global Stiffness matrix (K_global), the residual input vector (r_input), the 3 subdomains node vectors with their overlaps (idx), a list of the Multigird hierarchies (mg_hier) (which contains the information we need for each subdomain), and the same nPre, nPost, omega as Function 18. Inside it, it solves each subdomain system separately with the help of Function 15. and returns the final result (z).

But to have a working Multigrid we need to dive even further. Specifically, we need to build the Prolongation and Restriction operators which are used inside Function 15. (MG_Vcycle.jl) and when building the MG hierarchies (*Appendix E*) with the Gallerkin's Approach. In this Chapter we chose to construct the bilinear Prolongation operator (P_{op}) and then find the Restriction operator (R_{op}) from the transpose of P_{op}. The functions that build those operators are the following:

Function 19. *bilinearRnP.jl*

```
"""
    bracketing1D(val::Float64, sorted_lines::Vector{Float64})

This is a helper function made to create the 1D bracketing, i.e. to find the lines that bracket the fine
node and
output the Normalized Interpolation Coord xi

# Arguments
- **val::Float64 :** The fine node coordinate (x or y axis)
- **sorted_lines::Vector{Float64} :** The sorted coarse x or y lines

# Returns
- **idx1, idx2 :** The 2 coarse lines that bracket the fine node
- **xi :** The Normalized Interpolation Coordinate
```

```
"""
function bracketing1D(val::Float64, sorted_lines::Vector{Float64})
    len_lines = length(sorted_lines)
    if len_lines == 0
        return 0, 0, 0.0 # No lines to define a segment
    end
    if len_lines == 1 # Only one coarse line available
        # Return 1, 1 bracketing coarse lines &
        # Check if fine node (val) ≈ coordinate of the single coarse line -> return 0.0 (normalized pos)
        # Else check if val < coarse line's position -> if yes return 0.0, if not return 1.0
        return 1, 1, (isapprox(val, sorted_lines[1]) ? 0.0 : (val < sorted_lines[1] ? 0.0 : 1.0))
    end

    # Find k such that sorted_lines[k-1] <= val <= sorted_lines[k] (Bracketing)
    k_insert = searchsortedfirst(sorted_lines, val);
    idx1, idx2 = 0, 0; # Initialize 1st and 2nd bracketing coarse line indexes, respectively
    if k_insert == 1 # val <= sorted_lines[1] (at or before the very first coarse line)
        idx1 = 1;
        idx2 = 2; # Use the first segment [lines[1], lines[2]] for interpolation
    elseif k_insert > len_lines # val > sorted_lines[end] (after the very last coarse line)
        idx1 = len_lines - 1;
        idx2 = len_lines; # Use the last segment [lines[end-1], lines[end]] for interpolation
    else # sorted_lines[k_insert-1] < val <= sorted_lines[k_insert]
        if isapprox(val, sorted_lines[k_insert]) # val is on line[k_insert]
            idx1 = k_insert;
            idx2 = k_insert; # Coincident with this line
        else # val is between line[k_insert-1] and line[k_insert]
            idx1 = k_insert - 1;
            idx2 = k_insert;
        end
    end

    # If coincident, xi depends on which end of segment we consider it
    # If fine node is on coarse_lines[idx1], weights involve xi=0
    # If fine node is on coarse_lines[idx2], weights involve xi=1
    if idx1 == idx2 # Coincident with sorted_lines[idx1]
        # To form a segment for interpolation, if it's not the last point, use [idx1, idx1+1] with xi=0
        # If it is the last point, use [idx1-1, idx1] with xi=1
        if isapprox(val, sorted_lines[idx1])
            if idx1 < len_lines
                return idx1, idx1 + 1, 0.0 # Treat as start of segment [idx1, idx1+1]
            else
                return idx1 - 1, idx1, 1.0 # Treat as end of segment [idx1-1, idx1]
            end
        else # Should not happen if idx1==idx2 was due to coincidence (Error)
             return 0, 0, 0.0
        end
    end

    line_val1 = sorted_lines[idx1];
    line_val2 = sorted_lines[idx2];
    denominator = line_val2 - line_val1;

    if abs(denominator) < 1e-12 # Avoid division by zero
        # If val is also on this line, treat xi as 0 or 1 depending on which index it matched more
closely
        xi_norm = isapprox(val, line_val1) ? 0.0 : (isapprox(val, line_val2) ? 1.0 : 0.5) # Fallback if
lines too close
    else
        xi_norm = (val - line_val1) / denominator
    end

    return idx1, idx2, clamp(xi_norm, 0.0, 1.0) # Clamp to handle extrapolation robustly
end
```

```
"""
    build_P_bilinear_and_R_transpose(fine_coords::Matrix{Float64})

This function builds and returns the Prolongation and the Restriction operators and the list of fine
node indices that are also coarse nodes.

# Arguments
- **fine_coords::Matrix{Float64} :** All the fine (initial) coordinates of the grid

# Returns
- **R_op :** The Restriction operator
- **P_op :** The Prolongation operator
- **fine_indices_selected_as_coarse :** The list of fine node indices that are also coarse nodes
"""
function build_P_bilinear_and_R_transpose(fine_coords::Matrix{Float64})
    n_fine = size(fine_coords, 1);

    minNodes = 4 # Min fine nodes to attempt meaningful 2x2 coarse cell
    if n_fine < minNodes
        @warn "P-bilinear: Grid too small ($n_fine nodes). Returning identity operators."
        identity_op = sparse(1.0I, n_fine, n_fine);
        return identity_op, identity_op', collect(1:n_fine)
    end

    unique_x_fine = sort(unique(fine_coords[:, 1]));
    unique_y_fine = sort(unique(fine_coords[:, 2]));

    if length(unique_x_fine) < 2 || length(unique_y_fine) < 2
        @warn "P-bilinear: Not enough unique fine grid lines for coarsening. Returning identity."
        identity_op = sparse(1.0I, n_fine, n_fine);
        return identity_op, identity_op', collect(1:n_fine)
    end

    # Define Coarse Grid structure from fine_coords
    coarse_x_lines = unique_x_fine[1:2:end];
    coarse_y_lines = unique_y_fine[1:2:end];

    if length(coarse_x_lines) < 1 || length(coarse_y_lines) < 1 # Need at least one line
        @warn "P-bilinear: Coarsening produced no coarse grid lines. Returning identity."
        identity_op = sparse(1.0I, n_fine, n_fine);
        return identity_op, identity_op', collect(1:n_fine)
    end

    # Create a map from coarse (x,y) to its 1D local coarse index (1 to n_coarse)
    # And identify the original fine_coords indices that are these coarse nodes
    map_coarse_coord_to_local_idx = Dict{Tuple{Float64, Float64}, Int}();
    fine_indices_selected_as_coarse = Int[]; # Will store original fine indices


    temp_coarseCoords = Tuple{Float64,Float64}[]; # Coordinates of actual coarse nodes
    map_fine_coord_to_fine_idx = Dict((fine_coords[i,1], fine_coords[i,2]) => i for i=1:n_fine); # Fine
coords to idx mapping

    for y_val in coarse_y_lines
        for x_val in coarse_x_lines
            # Find fine nodes that are these coarse_x_lines/coarse_y_lines intersections
             fine_idx_match = get(map_fine_coord_to_fine_idx, (x_val, y_val), 0);
             if fine_idx_match != 0
                push!(temp_coarseCoords, (x_val,y_val));
             end
        end
    end

    # Re-sort temp_coarseCoords to ensure canonical order for map_coarse_coord_to_local_idx
    sort!(temp_coarseCoords, by = p->(p[2],p[1]));
```

```
    n_coarse = length(temp_coarseCoords);
    if n_coarse == 0 || n_coarse == n_fine # No actual coarse nodes formed or no coarsening
         @warn "P-bilinear: No effective coarse grid defined or no coarsening. Returning identity.";
        identity_op = sparse(1.0I, n_fine, n_fine);
        return identity_op, identity_op', collect(1:n_fine)
    end

    for i=1:n_coarse
        map_coarse_coord_to_local_idx[temp_coarseCoords[i]] = i;
        # Populate fine_indices_selected_as_coarse based on this sorted order
        push!(fine_indices_selected_as_coarse, map_fine_coord_to_fine_idx[temp_coarseCoords[i]]);
    end

    P_I, P_J, P_V = Int[], Int[], Float64[]; # Triplets for sparse P

    for i_fine = 1:n_fine
        xf, yf = fine_coords[i_fine, 1], fine_coords[i_fine, 2];

        ix1_line_idx, ix2_line_idx, xi_norm = bracketing1D(xf, coarse_x_lines);
        iy1_line_idx, iy2_line_idx, yi_norm = bracketing1D(yf, coarse_y_lines);

        # Define the 4 parent coarse cell corner coordinates
        parent_CornerCoords = [
            (coarse_x_lines[ix1_line_idx], coarse_y_lines[iy1_line_idx]), # c00
            (coarse_x_lines[ix2_line_idx], coarse_y_lines[iy1_line_idx]), # c10
            (coarse_x_lines[ix1_line_idx], coarse_y_lines[iy2_line_idx]), # c01
            (coarse_x_lines[ix2_line_idx], coarse_y_lines[iy2_line_idx])  # c11
        ];

        weights = [
            (1 - xi_norm) * (1 - yi_norm),  # for c00
            xi_norm       * (1 - yi_norm),  # for c10
            (1 - xi_norm) * yi_norm,        # for c01
            xi_norm       * yi_norm         # for c11
        ];

        total_weight = 0.0;
        active_parents_data = Tuple{Int, Float64}[]; # (coarse_local_idx, weight)

        for k_corner = 1:4
            coarseNodes = parent_CornerCoords[k_corner];
            weight = weights[k_corner];

            coarse_node_local_idx = get(map_coarse_coord_to_local_idx, coarseNodes, 0);
            if coarse_node_local_idx != 0 && weight > 1e-9 # If parent coarse node exists in the list
and weight is significant
                push!(active_parents_data, (coarse_node_local_idx, weight));
                total_weight += weight;
            end
        end

        if total_weight < 1e-9 # If no valid parents or all weights effectively zero
            min_d_sq = Inf; # Smallest squared distance so far
            best_c_local_idx = 0; # Stores the idx of the closest coarse node
            for i_c_search = 1:n_coarse
                c_coord = temp_coarseCoords[i_c_search];
                d_sq = (xf-c_coord[1])^2 + (yf-c_coord[2])^2; # Euclidean distance (squared distance)
                if d_sq < min_d_sq
                    min_d_sq = d_sq;
                    best_c_local_idx = i_c_search;
                end
            end
            if best_c_local_idx > 0
                push!(P_I, i_fine); push!(P_J, best_c_local_idx); push!(P_V, 1.0);
```

```
            end
            continue
        end

        # Normalize weights for this fine node to sum to 1
        for (coarse_idx, weight) in active_parents_data
            push!(P_I, i_fine);
            push!(P_J, coarse_idx); # This is the local coarse index (1 to n_coarse)
            push!(P_V, weight / total_weight) ;
        end
    end

    if isempty(P_I)
        @warn "P-bilinear: Prolongation operator P is empty. Returning identity.";
        identity_op = sparse(1.0I, n_fine, n_fine);
        return identity_op, identity_op', collect(1:n_fine)
    end

    P_op = sparse(P_I, P_J, P_V, n_fine, n_coarse);
    R_op = P_op'; # R = P^T

    return R_op, P_op, fine_indices_selected_as_coarse
end
```

The main function of this file is *build_P_bilinear_and_R_transpose()* while *bracketing1D()* is a helper function. All *build_P_bilinear_and_R_transpose()* needs for an input are the fine coordinates of the whole global grid (fine_coords). The approach begins by identifying a coarser grid from a given fine grid by sampling every other coordinate along each axis. Coarse nodes are selected from fine nodes that exactly match these sampled coordinates. Using the helper function, *bracketing1D*, each fine node is located within the appropriate coarse cell, and normalized distances are computed to evaluate bilinear interpolation weights relative to the surrounding four coarse nodes. The prolongation matrix P_{op} is then built using these weights, and its transpose provides the restriction operator R_{op}.

Using the codes from above we should have a working Multigrid Method and our Main now should look like this (for *MGHier_Setup()* refer to *Appendix E*):

Function 20. *Main.jl*

```
using Pkg
Pkg.activate(@__DIR__)
using MyFunctions, BenchmarkTools, SparseArrays, LinearAlgebra;
import PrettyTables;

step1 = 0.01;
GO = Lmesh(step1);
# Define the function as an anonymous function
s = 0.05;
k = 5.0;
f = (x, y) -> (-1/(2*pi*s^2)) * exp(-((x+0.5).^2 + (y-0.5).^2) / (2*s^2)) + (-1/(2*pi*s^2))
* exp(-((x-0.5).^2 + (y+0.5).^2) / (2*s^2));

println("Building global FEM matrices (k=$k)...");
```

```
@time "Global FEM Assembly" K1, F_global1, all_coords = QuadFEM_Matrices(GO, f, k);

H = 20; # Overlap
println("\nPerforming domain decomposition with overlap H = $H...")
@time "Subdomain Overlap" K_sub, F_sub, idx = SubdomainOverlap_Matrices(GO, K1, F_global1,
H);
println("Number of subdomains created: ", length(K_sub));
K_sub_factors = [factorize(K_s) for K_s in K_sub]; # Pre-factorize K_sub to improve
performance

# --- Multigrid Setup Parameters ---
num_mg_levels = 2; # Number of levels in hierarchy
min_coarse_size_param = 10; # Min nodes on coarsest grid for hierarchy building
# V-cycle parameters
nPre= 2; # Pre-smoothing iterations
nPost = 2; # Post-smoothing iterations
omega = 4.0/5.0; # Damping for Jacobi

# Setup the MG hierarchies
mg_hierarchies_list = MGHier_Setup(K_sub, idx, all_coords, num_mg_levels,
min_coarse_size_param);

# --- Solving the System ---
tol = 10^-4;
Nmax = 500; # Max iterations
krylov_dim_gmres = 30;

println("\n--- Solving with k=$k ---")

@time "Solving time" begin
    # u = Conj_Grad(K1, F_global1, tol, 1000);
    # u = pcdCG(K1, F_global1, tol, K_sub_factors, idx, 500);
    u = MGpcdCG(K1, F_global1, tol, idx, mg_hierarchies_list, Nmax, omega, nPre, nPost);
end
```

Now if we benchmark MGpcdCG against pcdCG and Conj_Grad we get the following results for $\mathrm{nPre} = \mathrm{nPost} = 2$ and $omega = \frac{4}{5}$ and 2 MG levels in the hierarchy.

Table 6

step = 0.01	Conj_Grad		pcdCG		MGpcdCG	
	Mean Time	Iterations	Mean Time	Iterations	Mean Time	Iterations
k=0.01, H=10	151.825 ms	217	41.190 ms	8	163.748 ms	12
k=0.01, H=20			41.691 ms	7	131.228 ms	9
k=0.01, H=50			47.399 ms	6	233.435 ms	12
k=5.0, H=10	213.816 ms	300	47.458 ms	9	199.657 ms	15
k=5.0, H=20			49.653 ms	8	190.466 ms	13
k=5.0, H=50			63.066 ms	8	384.751 ms	20
k=8.0, H=10	273.991 ms	373	59.925 ms	11	410.990 ms	31
k=8.0, H=20			71.185 ms	12	1.023 s	70
k=8.0, H=50			71.014 ms	9	424.561 ms	22

From Table 6 we derive that the best overall method, right now, is pcdCG but that is expected as the pre-factorization of each subdomain together with the "\" operator is extremely powerful. Furthermore, we see that MGpcdCG completely dominates Conj_Grad and is a close second to pcdCG in terms of iterations while it is also winning a lot of the times in terms of speed against Conj_Grad. The small number of iterations and the victories it already has against Conj_Grad suggest that with better optimization MGpcdCG will be very promising. Also, we must note that since, generally, the Conjugate Gradient method is not theoretically the correct choice all these 3 methods will probably be unstable for bigger problems, thus the implementation of Multigrid in GMRes might be beneficial for those cases.

Also, the observations we had about the balancing of the overlap value (H) in Chapter 7 are apparent here too. See, for example, in Table 6 the results of MGpcdCG for k = 0.01 where the iterations go from 12 to 9 and back to 12 again and the solving times from 163ms to 131ms and then 233ms while H increases from H = 10 to 20 and finally 40 respectively.

Appendix

A. Plot the L-Shaped Meshed Domain

Function 21. *PlotDomain.jl*

```
function PlotDomain(all_coords::Matrix{Float64})
    fig_mesh = GLMakie.Figure();

    # Plot domain with subdomain 1 overlaps
    ax_mesh = GLMakie.Axis(fig_mesh[1, 1], aspect = AxisAspect(1), xlabel =
"x", ylabel = "y",
                           title = "Physical Mesh Nodes of L-Shape Domain");

    GLMakie.scatter!(ax_mesh, all_coords[:,1], all_coords[:,2],
                     color = :blue, markersize = 5,
                     strokecolor = :black, strokewidth = 0.5, # Stroke is for
better visibility
                     );
    display(GLMakie.Screen(), fig_mesh);
end
```

B. Double Integral GQ Solver

Function 22. *DGQ_leg.jl*

```
"""
    DGQ_leg(f::Function, n::Integer)

# Arguments
- f::Function : The function f(x,y) you want to double integrate from -1 to 1 (This
calculates the Legendre GQ)
- n::Integer : The degree of the Gausian Quadrature. With n-point GQ you can solve for
polynomials of degree 2n-1 or less

# Returns
- res : The aproximate result of the Double Integration of f(x,y) from -1 to 1
"""
function DGQ_leg(f::Function, n::Integer) # Up to 5 point Double Gaussian Quadrature
(Legendre)

    # Gaussian Quadrature points and weights table
    GQ_Table = [
        ([0.0], [2.0]),
        ([+1/sqrt(3), - 1/sqrt(3)], [1.0, 1.0]),
        ([0, +sqrt(3/5), -sqrt(3/5)], [8/9, 5/9, 5/9]),
        ([+sqrt(3/7-(2/7)*sqrt(6/5)), -sqrt(3/7-(2/7)*sqrt(6/5)),
+sqrt(3/7+(2/7)*sqrt(6/5)), -sqrt(3/7+(2/7)*sqrt(6/5))]
            ,[(18+sqrt(30))/36, (18+sqrt(30))/36, (18-sqrt(30))/36, (18-sqrt(30))/36]),
        ([0, +(1/3)*sqrt(5-2*sqrt(10/7)), -(1/3)*sqrt(5-2*sqrt(10/7)),
+(1/3)*sqrt(5+2*sqrt(10/7)), -(1/3)*sqrt(5+2*sqrt(10/7))]
```

```
            ,[128/225, (322+13*sqrt(70))/900, (322+13*sqrt(70))/900, (322-
13*sqrt(70))/900, (322-13*sqrt(70))/900])
    ]
    # Points and Weights initialization
    p, w = GQ_Table[n];
    # Double Gaussian Quadrature calculation
    res = 0.0;
    for i = 1:n
        for j = 1:n
            res += + w[i] * w[j] * f(p[i], p[j]);
        end
    end
    return res
end
```

C. Plot the Solution

Function 23. *PlotSolutionAnimation.jl*

```
function PlotSolutionAnimation(GO::Vector{Matrix{Int64}}, u, k::Float64,
filename::String)
    GLMakie.activate!();
    rowNodes = length(GO[1][1,:]);
    newStep = 1/(rowNodes - 1); # Get step from the GO matrix
    c = 3*10^8; # Speed of the wave
    omega = k*c; # Angular Velocity
    T = 2*pi / omega; # Period
    frames = 60;
    ts = range(0, T, length=frames);  # Time frames

    # --- Mesh for plotting ---
    X1m, Y1m = ndgrid(-1:newStep:0, -1:newStep:0);
    X2m, Y2m = ndgrid(0:newStep:1, -1:newStep:0);
    X3m, Y3m = ndgrid(-1:newStep:0, 0:newStep:1);
    u1 = u[GO[1]]; U1 = reshape(u1, size(GO[1]));
    u2 = u[GO[2]]; U2 = reshape(u2, size(GO[2]));
    u3 = u[GO[3]]; U3 = reshape(u3, size(GO[3]));

    # --- Create Observable for each subdomain ---
    U1_t = Observable(real.(U1));
    U2_t = Observable(real.(U2));
    U3_t = Observable(real.(U3));

    # --- Create Figure ---
    fig2 = GLMakie.Figure();
    screen2 = display(GLMakie.Screen(), fig2);
    ax = GLMakie.Axis3(fig2[1,1], title = "Wave Propagation", xlabel="x",
ylabel="y", zlabel="u(x,y,t)");

    # --- Surface Plots -------------------------------------
    GLMakie.surface!(ax, X1m, Y1m, U1_t, colormap = :viridis);
```

```
    GLMakie.surface!(ax, X2m, Y2m, U2_t, colormap = :viridis);
    GLMakie.surface!(ax, X3m, Y3m, U3_t, colormap = :viridis);

    # --- Animation ----------------------------------------
    for t in ts
        phase = exp(1im * omega * t);
        U1_t[] = real.(U1 * phase);
        U2_t[] = real.(U2 * phase);
        U3_t[] = real.(U3 * phase);
        sleep(0.05)
    end
end
```

D. Plot Domain with Overlaps

Function 24. *PlotDomainOverlaps.jl*

```
function PlotDomainOverlaps(all_coords::Matrix{Float64}, idx::Vector{Vector{Int64}})

    fig_mesh = GLMakie.Figure();

    # Plot domain with subdomain 1 overlaps
    ax_mesh = GLMakie.Axis(fig_mesh[2, 1:2], aspect = AxisAspect(1), xlabel = "x",
ylabel = "y",
                           title = "Physical Mesh Nodes of L-Shape Domain with Subdomain
1 Overlaps");

    GLMakie.scatter!(ax_mesh, all_coords[:,1], all_coords[:,2],
                     color = :blue, markersize = 5,
                     strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                     );

    GLMakie.scatter!(ax_mesh, all_coords[idx[1],1], all_coords[idx[1],2],
                     color = :red, markersize = 4,
                     strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                     );

    # Plot domain with subdomain 2 overlaps
    ax_mesh2 = GLMakie.Axis(fig_mesh[1,1], aspect = AxisAspect(1), xlabel = "x", ylabel
= "y",
                            title = "Physical Mesh Nodes of L-Shape Domain with
Subdomain 2 Overlaps");

    GLMakie.scatter!(ax_mesh2, all_coords[:,1], all_coords[:,2],
                     color = :blue, markersize = 5,
                     strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                     );

    GLMakie.scatter!(ax_mesh2, all_coords[idx[2],1], all_coords[idx[2],2],
```

```
                    color = :red, markersize = 4,
                    strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                    );

    # Plot domain with subdomain 3 overlaps
    ax_mesh3 = GLMakie.Axis(fig_mesh[1, 2], aspect = AxisAspect(1), xlabel = "x", ylabel
= "y",
                            title = "Physical Mesh Nodes of L-Shape Domain with
Subdomain 3 Overlaps");

    GLMakie.scatter!(ax_mesh3, all_coords[:,1], all_coords[:,2],
                    color = :blue, markersize = 5,
                    strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                    );

    GLMakie.scatter!(ax_mesh3, all_coords[idx[3],1], all_coords[idx[3],2],
                    color = :red, markersize = 4,
                    strokecolor = :black, strokewidth = 0.5, # Stroke is for better
visibility
                    );

    display(GLMakie.Screen(), fig_mesh);
end
```

E. Building and Setting Up the Multigrid Hierarchies

Function 25. *build_mg_hierachies.jl*

```
function build_mg_hierarchy(
    A_fine::SparseMatrixCSC{Float64, Int64},
    coords_arg::Matrix{Float64},
    nodes_arg::Vector{Int},
    min_size::Int=5,
    max_levels::Int=5
    )

    levels = MGLevel[];
    current_A = A_fine;
    current_coords = coords_arg;
    current_nodes = nodes_arg;

    for _ in 1:max_levels

        if size(current_A, 1) <= min_size
            println("Current A size ($(size(current_A,1))) <= min_size ($min_size).
Stopping coarsening.");
            break
        end

        # Build injection Restriction & Prolongation operators
        R, P, coarse_indices = build_P_bilinear_and_R_transpose(current_coords);
```

```
        nc = size(R, 1) # Number of coarse nodes for this new level

        if nc <= min_size
            println("Number of coarse nodes ($nc) <= min_size ($min_size) after
injection. Stopping coarsening.");
            break
        end

        # Galerkins coarse operator
        A_coarse = R * current_A * P;

        # Store current level's A and R, P
        push!(levels, MGLevel(current_A, R, P, current_nodes, current_coords));

        # Prepare next level
        current_A = A_coarse;
        current_coords = current_coords[coarse_indices, :];
        current_nodes = current_nodes[coarse_indices];
    end

    # Add coarsest level
    push!(levels, MGLevel(current_A,
                          nothing,  # R = nothing
                          nothing,  # P = nothing
                          current_nodes,
                          current_coords));

    return MGHier(levels)
end
```

Function 26. *MGHier_Setup.jl*

```
function MGHier_Setup(
    K_sub::Vector{SparseMatrixCSC{Float64, Int64}},
    idx::Vector{Vector{Int64}},
    all_coords::Matrix{Float64},
    num_mg_levels::Int,
    min_coarse_size_param::Int
    )
    println("\n--- Setting up Multigrid Hierarchies for All Subdomains ---")
    mg_hierarchies_list = Vector{MGHier}(undef, length(K_sub)) # To store MGHier for
each subdomain

    for i = 1:3
        println("  Processing Subdomain $i for MG setup...");
        A_sub_i = K_sub[i];
        subdomain_global_indices = idx[i];

        if isempty(A_sub_i) || isempty(subdomain_global_indices)
            @warn "  Subdomain $i: A_sub or node list is empty. Cannot build hierarchy."
            # Create an MGHier with no levels, or a single level if A_sub_i exists
            mg_hierarchies_list[i] = MGHier(MGLevel[]) # Empty hierarchy
            continue
        end
```

```
        # Get coordinates for the current subdomain's nodes
        sub_coords_i = all_coords[subdomain_global_indices, :]

        mg_hierarchies_list[i] = build_mg_hierarchy(
            A_sub_i,
            sub_coords_i,
            copy(subdomain_global_indices), # Pass a copy of original global indices for
this subdomain
            min_coarse_size_param,
            num_mg_levels
        );

        if isempty(mg_hierarchies_list[i].levels)
            @warn "  Subdomain $i: MG hierarchy is empty after build."
        else
            println("  Subdomain $i: Stored $(length(mg_hierarchies_list[i].levels)) MG
levels.")
        end
    end
    println("--- Finished Multigrid Hierarchies Setup ---");
    return mg_hierarchies_list;
end
```

References

- Saad Yousef Iterative, Methods for Sparse Linear Systems
- Anastasis C. Polycarpou, Introduction to the Finite Element Method in Electromagnetics
- William L. Briggs, Van Emden Henson, Steve F. McCormick, A Multigrid Tutorial, 2nd Edition
- Volker John, Multigrid Methods, Winter Semester 2013/14
- Dr. Cüneyt Sert, ME 582 Finite Element Analysis in Thermofluids, Chapter 3
- Pen State University, The Preconditioned Conjugate Gradient Method, Lecture #20
- Caraba, Elena, "PRECONDITIONED CONJUGATE GRADIENT ALGORITHM" (2008). Honors Theses. 269.
- https://en.wikipedia.org/wiki/LU_decomposition
- https://en.wikipedia.org/wiki/Generalized_minimal_residual_method
- https://en.wikipedia.org/wiki/Schwarz_alternating_method
- https://en.wikipedia.org/wiki/Conjugate_gradient_method
- https://www.cse.psu.edu/~b58/cse456/lecture20.pdf
- Πολυπλεγματικές Μέθοδοι, ECE DUTH eclass https://eclass.duth.gr/modules/document/file.php/TMA399/%CE%9A%CE%95%CE%A6%CE%91%CE%9B%CE%91%CE%99%CE%9F_MULTIGRID_v1.pdf
- https://courses.grainger.illinois.edu/cs357/sp2022/notes/ref-9-linsys.html
- https://docs.julialang.org
- https://math.mit.edu/classes/18.086/2006/am63.pdf