Interface Numerical Method for Coupling Independently Modeled Substructures

August 23, 2004

Dr. Padmanabhan Seshaiyer
Faculty Mentor

Christina Anaya
McNair Scholar

ABSTRACT

In many real world applications, especially in engineering, solutions over large complex structures and/or domains are required. By dividing the domain into substructures, solving the problem on each independent region, and then combining the independent solutions to analyze the entire region, the solution to the global domain can be found efficiently. In this paper, I will discuss background research in this field, illustrate examples of different methods that have been used, formulate an interface numerical method that can be employed for coupling independently modeled substructures, and outline a solution methodology for solving such problems.

1. INTRODUCTION

In the world today, teamwork has become the norm. Thus as applications are divided into separate components among different people or companies, a problem arises when the components must be assembled. Numerical methods using finite elements allow the combination of such separate component models in an efficient manner.

For example, one may be interested in studying a physical process on the large domain Ω (see Figure 1).

Figure 1. Global domain

To simplify the solution methodology, the global domain Ω can be divided into two separate subdomains, Ω_1 and Ω_2 (see Figure 2).

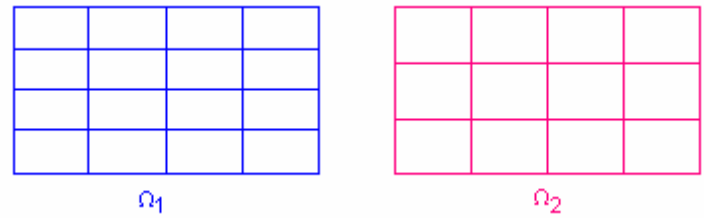

Figure 2. Global domain divided into two subdomains

The subdomains may be analyzed by independent modelers and hence may possess different mesh properties. This causes problems when trying to integrate the two due to the non-conformity at the interface (see Figure 3).

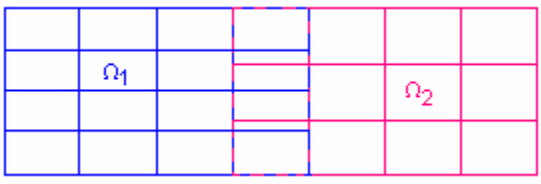

Figure 3. Assembled global domain showing overlapping interface

To study a physical system, a *mathematical model* is often constructed. To solve the associated mathematical model one may employ one of the two types of solution methodologies: *analytical* and *numerical*. An *analytical solution* is an exact answer to a problem. On the other hand, a *numerical solution* is just an approximation. The word "approximation" may cast doubt in people's mind; however there are numerous mathematical methods that demonstrate that an "approximate" answer can be found which estimates the analytical solution within the precision available on today's computer systems. A model flowchart of the process is described in Figure 4.

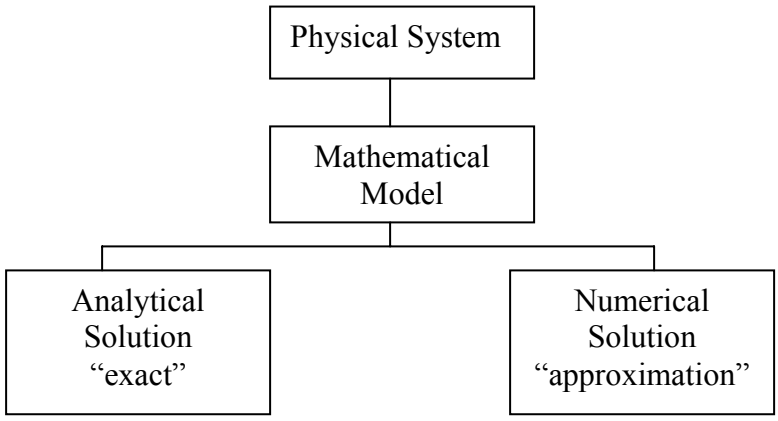

Figure 4. Solution methodology

If the mathematical model is simple, one can obtain an analytical solution. Most real world problems, however, are much more complex and hence require a numerical solution. In this paper, one such complex application is examined. The numerical methods that were used in this project were the finite difference method, the Jacobi iterative method, and spline interpolation techniques for solving a model diffusion equation on a complex domain.

2. BACKGROUND AND METHODS

In this section, we describe the techniques that will be employed later for one dimensional and two dimensional model problems.

2.1 A One Dimensional Model Problem

Consider the following physical system consisting of an elastic bar under stress via a tangential force F(x) with fixed endpoints *a* and *b* (see Figure 5). The objective is to determine the displacement U(x).

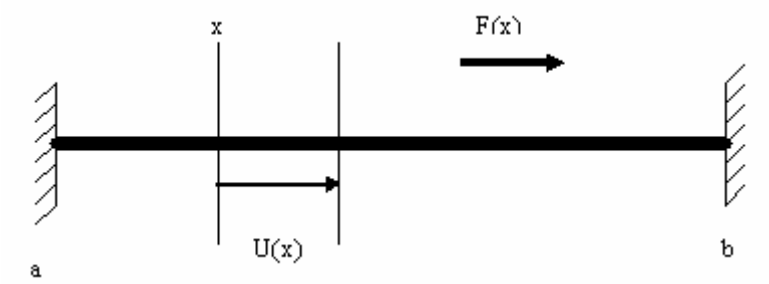

Figure 5. Elastic bar

The mathematical model can be derived as follows. First recall Hooke's Law, $\sigma = E\,\frac{dU}{dx}$. As demonstrated in [4], Hooke's law states that stress is directly proportional to strain, where E is a constant of proportionality known as *Young's Modulus*. For simplicity we let $E = 1$ and the equation becomes $\sigma = \frac{dU}{dx}$. Taking the derivative of both sides yields $\frac{d\sigma}{dx} = \frac{d^2U}{dx^2}$. Moreover, the equilibrium of the elastic bar system yields $\frac{d\sigma}{dx} = -F(x)$. The model then becomes the following boundary value problem (BVP),

$$-\frac{d^2U}{dx^2} = F(x),\ a < x < b$$

$$U(a) = 0$$
$$U(b) = 0$$

where *U(a) = 0* and *U(b) =0* are considered the boundary conditions which are needed to solve the differential equation uniquely. To find *U(x)* numerically, first divide the interval [*a, b*] into *N* subintervals with the nodes $a = x_0, x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_N = b$. Then rewrite the continuous differential equation on each subinterval to give a discrete differential equation by using the finite difference method as follows,

$$\frac{d^2U}{dx^2} = \frac{d}{dx}\left(\frac{dU}{dx}\right) = \frac{dg}{dx} = \frac{g(x_{i+1}) - g(x_i)}{\Delta x}$$

$$= \frac{\frac{dU}{dx}(x_{i+1}) - \frac{dU}{dx}(x_i)}{\Delta x}$$

$$= \frac{\frac{U(x_{i+1}) - U(x_i)}{\Delta x} - \frac{U(x_i) - U(x_{i-1})}{\Delta x}}{\Delta x}$$

$$\therefore \quad \frac{d^2U}{dx^2} \cong \frac{U(x_{i+1}) - 2U(x_i) + U(x_{i-1})}{\Delta x^2}$$

Since $\frac{d^2U}{dx^2} = -F(x)$, the equation becomes

$$\frac{U(x_{i+1}) - 2U(x_i) + U(x_{i-1})}{\Delta x^2} = -F(x).$$

Imposing the discrete equation for i = 2, 3, …., (*n-1)* and considering the given boundary conditions, the result is the following linear system of equations.

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \ldots\ldots\ldots & 0 \\ 1 & -2 & 1 & 0 & 0 & \ldots\ldots\ldots & 0 \\ 0 & 1 & -2 & 1 & 0 & \ldots\ldots\ldots & 0 \\ 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 \\ 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 \\ 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 & 1 & -2 & 1 \\ 0 & \ldots\ldots\ldots & \ldots\ldots\ldots & 0 & 1 & -2 & \end{bmatrix} \begin{bmatrix} U(x_2) \\ U(x_3) \\ U(x_4) \\ \ldots. \\ \ldots. \\ U(x_{n-2}) \\ U(x_{n-1}) \end{bmatrix} = -\Delta x^2 \begin{bmatrix} F(x_2) \\ F(x_3) \\ F(x_4) \\ \ldots. \\ \ldots. \\ F(x_{n-2}) \\ F(x_{n-1}) \end{bmatrix}$$

The linear system can be represented as $AU = F$, where A is the tridiagonal matrix, and U and F are the vectors at each point x_i. This can be solved with Matlab using the command $U = A \setminus F$.

Note that the finite difference method can also be used to solve more general boundary value problems, such as [5]

$$y'' + P(x)y' + Q(x)y = f(x),$$

with y as the dependent variable and subject to the conditions $y(a) = y_0, y(b) = y_1$. The finite difference equation would be:

$$h^2 f_i = (1 + \frac{h}{2} P_i) y_{i+1} + (-2 + h^2 Q_i) y_i + (1 - \frac{h}{2} P_i) y_{i-1},$$

where $h = \frac{(b-a)}{n}, P_i = P(x_i), Q_i = Q(x_i), f_i = f(x_i), y_i = y(x_i)$. The difference equation would be used to calculate the numerical solution in a similar fashion as the model problem.

Also note that the matrix A has a lot of zero entries which makes it a *sparse* matrix. An iterative technique for solving large systems with a large number of zero entries is the Jacobi Method. As is explained in [1], "to solve the linear system **Ax = b**" requires an initial approximation $\mathbf{x}^{(0)}$ to the solution **x**, which "generates a sequence of vectors that converges to **x**." With the Jacobi Iterative Method, the system **Ax = b** is converted into an equivalent system of the form $\boldsymbol{x} = T\,\boldsymbol{x} + \boldsymbol{c}$, where T is a fixed matrix and $\boldsymbol{c}$ is a vector. The initial approximation $\mathbf{x}^{(0)}$ is chosen and the solution vector is generated by computing

$$\boldsymbol{x}^{(k)} = T\,\boldsymbol{x}^{(k-1)} + \boldsymbol{c},$$

for k = 1, 2, 3,... Iterative techniques are efficient when considering computational time and computer storage [1].

2.2 A Two Dimensional Model Problem

Similar physical systems, as in 1-D, can be modeled using a two dimensional representation. An example of a 2-D model is the temperature distribution in a slab [5] (see Figure 6.)

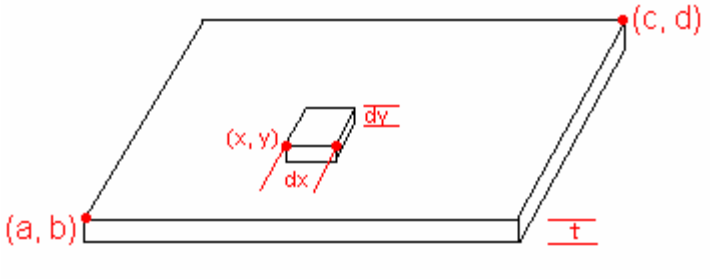

Figure 6. Thin slab

This is a slab of uniform material and thickness, t. If the temperature within each element of size dx X dy is represented with the dependent variable, U, then U_{xy} is the temperature at each x-y point of the slab. An extension of the model introduced in § 2.1 to 2-D can be given by

$$
\begin{aligned}
-U_{xx} - U_{yy} &= F(x,y), \quad x \in (a,b) \times (c,d) \\
U(a,y) &= U_{ay}, \qquad b < y < d \\
U(x,b) &= U_{xb}, \qquad a < x < c \\
U(c,y) &= U_{cy}, \qquad b < y < d \\
U(x,d) &= U_{xd}, \qquad a < x < c.
\end{aligned}
$$

One can discretize this model over the domain Ω (see Figure 7).

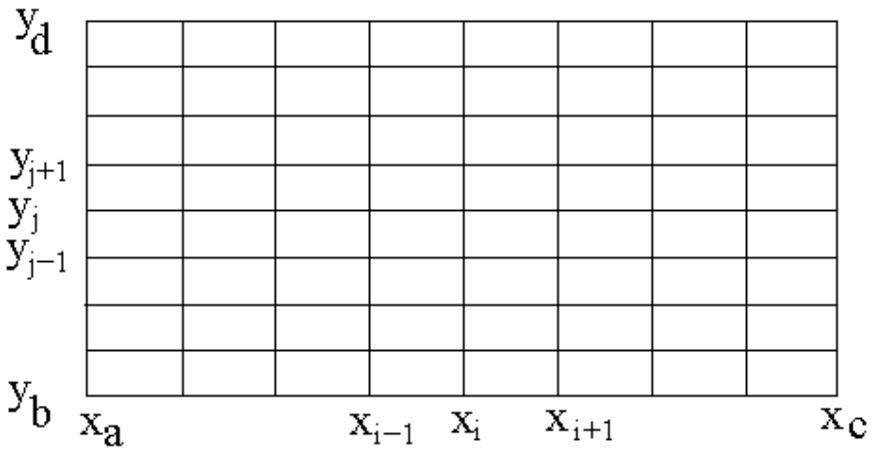

Figure 7. Discretization of domain Ω

using finite differences again,

$$
\frac{U(x_{i+1},y_j) - 2U(x_i,y_j) + U(x_{i-1},y_j)}{\Delta x^2} + \frac{U(x_i,y_{j+1}) - 2U(x_i,y_j) + U(x_i,y_{j-1})}{\Delta y^2} = -F(x_i,y_j).
$$

If the step size in the x-direction equals the step size in the y-direction, i.e. $\Delta x = \Delta y$, the equation becomes

$$\frac{U(x_{i+1}, y_j) + U(x_{i-1}, y_j) + U(x_i, y_{j+1}) + U(x_i, y_{j-1}) - 4U(x_i, y_j)}{\Delta x^2} = -F(x_i, y_j).$$

To simplify notation, each point (x, y) is represented by a single number called a node. For example, (1, 1) is node 15 and (7, 3) is node 7 as shown in Figure 8.

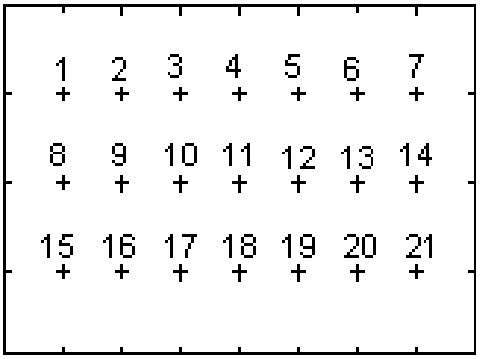

Figure 8. Nodal numbering

The discretized system of equations becomes:

$$\begin{bmatrix}
-4 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\cdots\cdots & & & & & & & & & & & & & & & & & & & & \\
\cdots\cdots & & & & & & & & & & & & & & & & & & & & \\
\cdots\cdots & & & & & & & & & & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -4
\end{bmatrix}
\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ \cdots \\ \cdots \\ U_{20} \\ U_{21} \end{bmatrix}
= \Delta h^2
\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ \cdots \\ \cdots \\ F_{20} \\ F_{21} \end{bmatrix}$$

where $\Delta h = \Delta x = \Delta y$ is the space between each node.

2.3 Interpolation

When data over a large domain is required and only minimal data points are available, the result may exhibit erratic behavior between the points. Cubic spline interpolation computes intermediate values between each data point [3]. If data is known at specific values, say x = 1, 2, 3, a reasonable estimation of x at 1.5 and 2.5 can be found through interpolation. In Matlab, YY = SPLINE(X, Y, XX) uses cubic spline interpolation to find a vector YY corresponding to XX which is the x value vector of the data points for which the estimation is

required. X and Y are the given data vectors [3]. For example, as shown in Figure 9, if X = *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10* and Y = *sin(X)*, the graph is

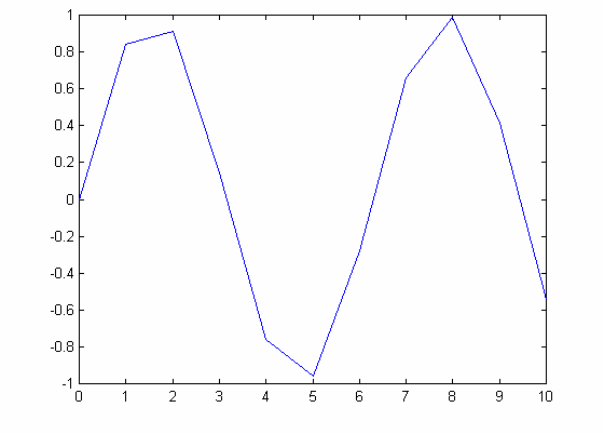

Figure 9. Graph of Y = *sin(X)*

Now, if spline interpolation is used and XX is defined as [0:0.5:10], the graph is much smoother (see Figure 10).

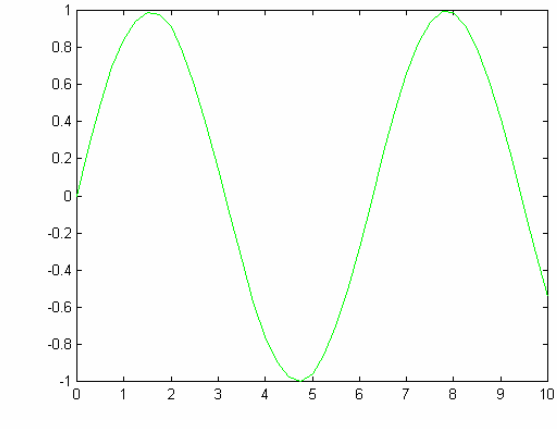

Figure 10. Graph of YY = sin(XX) using spline interpolation

The interpolation allows for the refinement of the YY values for more points in the domain.

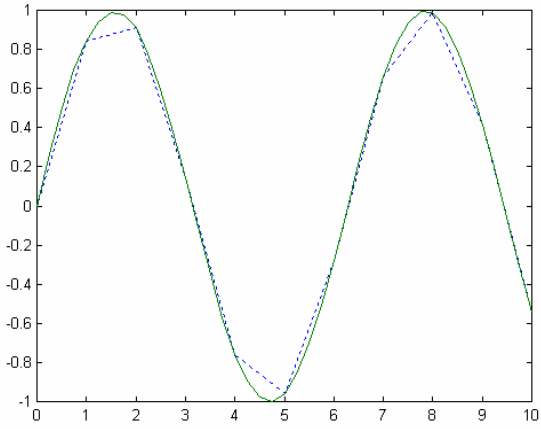

Figure 11. Graph of Y = *sin (X)* in blue and graph of YY using spline interpolation in green

3. THE INTERFACE PROBLEM

In this section, we will describe the interface problem and formulate an overlapping interface technique to solve it.

3.1 Division of Domain

Consider a global domain Ω divided into two subdomains, Ω_1 and Ω_2, with $m = 4$ and $n = 3$, where m and n represent the number of rows in each subdomain (see Figures 12 & 13). Let us for simplicity assume that both subdomains have equal number of columns. The solution is computed independently for Ω_1 and Ω_2 using the previously discussed methods.

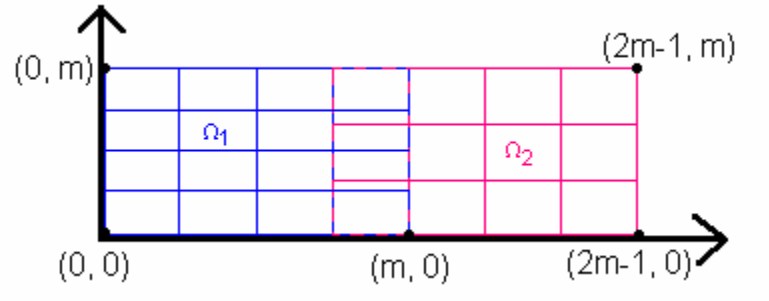

Figure 12. Global domain in x-y plot

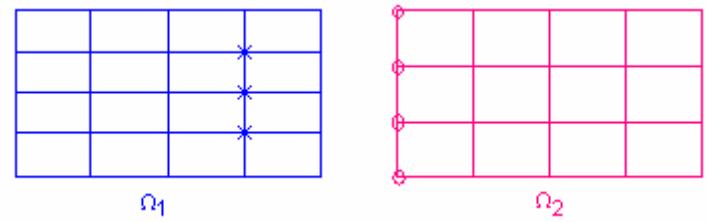

Figure 13. Subdomains, Ω_1 and Ω_2, with $m = 4$ and $n = 3$, respectively

Once the subdomains are combined, the individual nodes on the interface do not align. A magnified view of the interface can be seen in Figure 14.

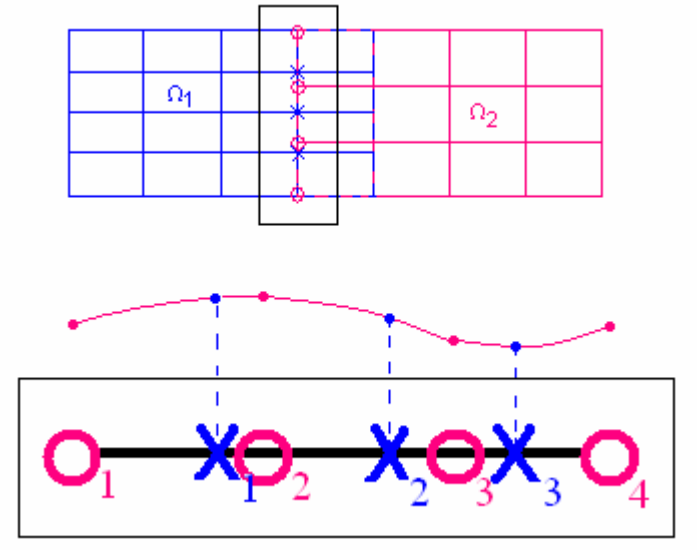

Figure 14. Overlapping interface magnified

With the use of cubic spline data interpolation, an interface numerical solution is constructed, which serves as a boundary condition for each iteration, as described next.

3.2 Methodology

The code for the overlapping method, found in the appendix, allows for a mesh of any size to be constructed by passing the number of nodes and the number of columns in which the nodes will lie. In Ω_1 of Figure 12, the number of nodes and number of columns in which the nodes lie are 25 and 5, respectively, so the parameters are (25, 5) . The program constructs the A matrix and allows the user to enter F and the boundary conditions, as examined in § 2.1. Once an initial U_1 vector is found, data from that vector is used for the interpolation. Data calculated using the spline interpolation is then used to solve the problem on Ω_2. The same method is followed in the second domain with new boundary values. U_2 is found for this domain and used for interpolation. The process is continued until the global solution is within a predefined tolerance. Next, we describe a pseudo code for the overlapping method algorithm.

3.3 Psuedo Code

Do

UU1 = Jacobi (n1, c1, f1, U_1)

n1 = number of nodes in first domain

c1 = number of columns in which the nodes will lie

f1 = F vector

U_1 = U vector (initial guess $U_1^{(0)}$)

UU1 = new U_1 which is returned

YY = SPLINE(X, Y, XX)

X = [0:1:(n1/c1-1)]

Y = the vector constructed from the UU1 results that correspond with the left boundary of the second domain

XX = [0:(n1/c1-1)/(n2/c2-1):(n1/c1-1)]

YY = the left boundary condition of the second domain

Use YY as a right boundary to determine U_2

UU2 = Jacobi (n2, c2, f2, U_2)

n2 = number of nodes in second domain

c2 = number of columns in which the nodes will lie

f2 = F vector

U_2 = U vector

UU2 = new U_2 which is returned

Y = SPLINE(XX, YY, X)

XX = [0:(n1/c1-1)/(n2/c2-1):(n1/c1-1)]

YY = the vector constructed from the UU2 results that correspond with the right boundary of the first domain

X = [0:1:(n1/c1-1)]

Y = the new right boundary of the first domain

Use Y as a left boundary to determine U_1

While (max number of iterations or predefined tolerance has not been reached)

4. FUTURE RESEARCH

When the large global domain is divided into several subdomains, each local domain can be analyzed independently. However, in some instances, information calculated in one local domain is required to calculate the solution of another. If each local domain is working on different processors, the sharing of such information would be easy. Message Passing Interface, MPI, is a library specification "designed for high performance on parallel machines and workstation clusters [2]." Information would be shared between processors or each of the domains allowing for ease of reconstruction of the global domain. We propose to implement the solution methodology developed herein in parallel using a MPI algorithm.

5. ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Padmanabhan Seshaiyer, my faculty mentor, for his guidance, dedication, and expertise. I would also like to thank Mr. Edward Swim, my graduate student mentor, for assisting me throughout the school year with his knowledge of graduate school and research.

Special thanks to the McNair Scholars' Program for supporting my research work and the National Science Foundation project under grant DMS0207327 awarded to Dr. Seshaiyer for motivating this problem.

6. REFERENCES

[1] Burden, Richard L. & Faires, J.D., *Numerical Analysis*, Brooks/Cole Publishing Company, 1997.

[2] *The Message Passing Interface (MPI)*, http://www-unix.mcs.anl.gov/mpi/

[3] Palm, William J., *Matlab for Engineering Applications*, McGraw-Hill, 1999.

[4] Serway, Raymond A. & Beichner, Robert J., *Physics: For Scientists and Engineers*, Saunders College Publishing, 2000.

[5] Zill, Dennis G. & Cullen, Michael R., *Differential Equations with Boundary-Value Problems*, Brooks/Cole Publishing Company, 1997.

7. APPENDIX

```
%%%%%%%%%%%%%%%%%%%%%
%   interface.m                                 %
%%%%%%%%%%%%%%%%%%%%%%

max = 0;      %max number of iterations

n1 = input('Please enter the number of nodes in the first domain.  ')
c1 = input('Please enter the number of columns in which the nodes will lie. ')
disp('Please enter the vector F in the form [1;2;3;4]')
f1 = input('F = ')
disp('Please enter your initial condition U(0)in the form [0;0;0;0]')
u1 = input('U(0) = ')

n2 = input('Please enter the number of nodes in the second domain.    ')
c2 = input('Please enter the number of columns in which the nodes will lie  ')
disp('Please enter the vector F in the form [1;2;3;4]')
f2 = input('F = ')
disp('Please enter your initial condition U(0)in the form [0;0;0;0]')
u2 = input('U(0) = ')

x=[0:1:((n1/c1)-1)];         %x vector constructed to use cubic spline interpolation
xx = [0:((n1/c1)-1)/((n2/c2)-1):((n1/c1)-1)]; %xx vector constructed to use cubic spline interpolation

while (max < 100)

      UU1 = Jacobi(n1, c1, f1, u1);      %Jacobian Method on first domain

      for i=1:1:(n1/c1)
       y(i,1)=UU1((n1-((i-1)*c1)-1),1);  %y vector related to x vector
      end

      yy = SPLINE(x, y, xx);    %yy vector found using spline interpolation

      for i=1:1:(n2/c2)
      u2((1+n2-(i*c2)),1)= yy(i,1);  %yy vector is new left boundary of second domain
      end

      UU2 = Jacobi(n2, c2, f2, u2);      %Jacobian Method on second domain

      for i=1:1:(n2/c2)
       yy(i,1)=UU2((n2-(i*c2)+2),1);     %yy vector related to xx vector
      end

      y  = SPLINE(xx, yy, x);   %y vector found using spline interpolation

      for i=0:1:((n1/c1)-1)
      u1(n1-(i*c1),1)=y(i+1,1); %y vector is new right boundary of first domain
      end

      max = max+1;
end
UU1     %display UU1 and UU2 results
UU2

%%%%%%%%%%%%          end interface.m          %%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%
%   jacobi.m                                %
%%%%%%%%%%%%%%%%%%%%%

function UU = jacobi(n, c, f, u)

for i=1:1:n
   k(i,1)=-4;
end

A= diag(k,0)+ diag(ones((n-1),1), -1) + diag(ones((n-1),1), 1) +diag(ones((n-c),1),c)+
                  diag(ones((n-c),1),-c);

for i=1:1:((n/c)-1)
   A(i*c,(i*c+1))=0;
   A((i*c+1),i*c)=0;
end

tolerance = 0.000001;

for i=1:n                    % Constructing the diagonal matrix D
  D(i,i)= A(i,i);
end;

for i=1:n                % Constructing the lower diagonal matrix L
  for j=1:n              % and the upper diagonal matrix U
    if j < i
      L(i,j) = -A(i,j);
      U(i,j) = [0];
    else
      L(i,j) = [0];
      if j > i
        U(i,j) = -A(i,j);
      else
        U(i,j) = [0];
      end;
    end;
  end;
end;

 %Jacobi
    T = inv(D) * (L + U);
    C = inv(D) * f;

error=1;
count=0;
while (error > tolerance)                % will continue to calculate until within tolerance
   answer=T*u + C;
   error=norm(answer - u, inf);
   count=count+1;
   u=answer;
end;
UU=u;

%%%%%%%%%%%%          end jacobi.m          %%%%%%%%%%%%%%%%
```