1. *Determine the expressions for the mapping functions, x(ξ, η) and y(ξ, η ). Indicate how you will use your programming language (MATLAB) to solve for the values of the coefficients of the analytical mapping. Demonstrate that your mapping works by showing the mapping of points between the computational and the physical space.*

The physical and computational domains are specified as:

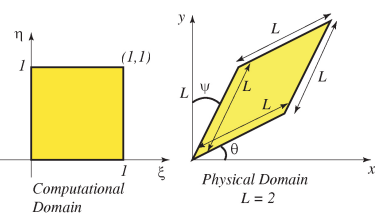


Figure 1: Domain specifications and parameters [1]

A bilinear mapping is used of the form:

*x(ξ, η) = sxξη + txξ + uxη + vx* (1a)

*y(ξ, η) = syξη + tyξ + uyη + vy* (1b)

Where the s, t, u, v coefficients must satisfy compatibility between domain vertices as described in Equations 2a and 2b. These are then solved using Matlab’s backslash command.

(2a)

(2b)

To verify correct mapping 10 random test points were chosen in the computational domain and then mapped to the physical domain. A figure was generated of the two domains and a visual comparison corroborated expected behavior. Figure 2 presents an example plot. Note the physical domain was manually offset for visibility, otherwise overlapping features obscure some detail.



Figure 2: Verification of mapping function

*2. Write the second order finite difference equations that will be used to represent the partial differential equation in the computational domain. Draw the finite difference stencil that will be used for stamping the values into the matrix and write the coefficients for each point on the finite difference stencil. Indicate how values from your stencil are going to be stamped into the matrix.*

The general PDE that describes the membrane deflection is:

(3)

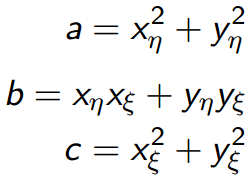
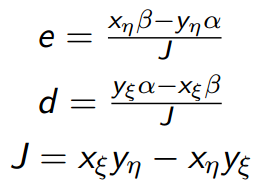
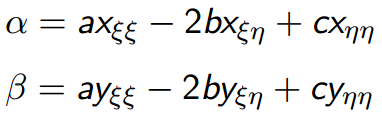
Specifically in the physical domain:

*-(uxx + uyy) = f* (4)

However this must be computed in the computational domain. Performing some additional manipulation on Equation 4 permits an expression in terms of the derivatives in the computational domain:

-1/J² (au*ξξ – 2buξη + cuηη + duη +euξ) = F* (5)

Where the coefficients a, b, c, d, e are defined in terms of spatial derivatives of the mapping functions (Equations 1a and 1b) in the computational domain [1]:

  (6)

The 2nd order accurate finite spatial derivatives in the computational domain are as follows:

u*ξξ = (ui+1,j -2ui,j +ui-1,j)/(Δξ)²* (7a)

u*ξη = (ui+1,j+1 -ui+1,j-1 -ui-1,j+1+ui-1,j-1)/(4ΔηΔξ)* (7b)

u*ηη = (ui,j+1 -2ui,j +ui,j-1)/(Δη)²* (7c)

u*η = (ui,j+1 -ui,j-1)/(2Δη)* (7d)

u*ξ = (ui+1,j -ui-1,j)/(2Δξ)* (7e)

Substituing Equations 7a-e and 6 into Equation 5, using the mapping coefficients obtained from Equations 1 and 2 yields the stencil to be used in this case (the common factor to all terms -1/J² is left out for brevity):

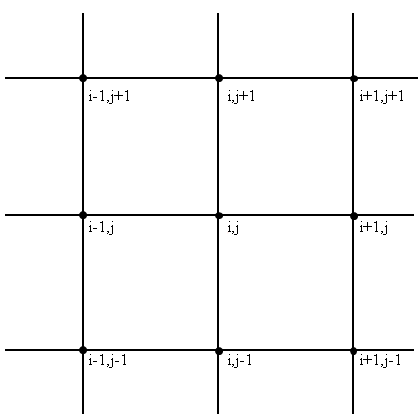


Figure : Finite difference stencil for computational domain

|  |  |  |
| --- | --- | --- |
| 2b/(4ΔηΔξ) | c/(Δ*η*)² +d/(2Δ*η*) | -2b/(4ΔηΔξ) |
| a/(Δξ)² -e/(2Δξ) | -2a/(Δξ)² - 2c/(Δη)² | a/(Δξ)² +e/(2Δξ) |
| -2b/(4ΔηΔξ) | c/(Δ*η*)² -d/(2Δ*η*) | 2b/(4ΔηΔξ) |

The nodes of the membrane are numbered sequentially starting at the origin and proceeding in increasing i, and then j. Thus any node’s number is simply:

N = *i+ [(number of nodes along i)\*(j-1)]* (8)

Each node to be solved ui,j will be a member in the vector with position uN. Likewise each of the stencil values will be located in its appropriate column N in the “A” matrix depending on its i,j location in the node’s respective stencil.

Of particular note is that because of the configuration of the physical domain, the mapping coefficients are constant across the entire domain, allowing potential simplification of the code.

*3. Write pseudo-code that describes the Matlab process of setting up and solving the finite difference problem.*

Set-up computational domain  
Set-up physical domain  
Calculate bilinear mapping coefficients  
Calculate sequential node number from i,j location  
Calculate transform equation coefficients  
Set-up ‘A’ matrix using transform coefficients and finite difference stencil  
Set-up force vector  
Apply BCs  
Solve matrix  
Plot  
  
*4. Using the provided Matlab skeleton code (or by writing your own version), fill in the missing*

*parts so that the code solves the deflection of a membrane on the non-rectangular domain.*

Using the pseudo-code from question 3 and the provided skeleton code, a full version of the mapper and solver was implemented. Full code can be found in Appendix A.

Figures 4 and 5 provide an example solution for θ=15°, φ=10° and N=33 for the computational and physical domain solutions respectively.



Figure : Computational domain solution.



Figure 5: Physical domain solution.

*5. When the angle* θ *and* φ *are both equal to zero, compute the convergence of the error of the maximum deflection of the membrane. Illustrate that the convergence is as expected using an appropriate log-plot. Use a 65x65 nodes solution as the "actual" deflection of the membrane.*

Figure 6 is a log-log plot of the convergence rate of the numerical solution. Not directly evident from the graph is that the computed slope is for the linear region farther away from the side node number of the solution used for comparison. For grid sizes near that of the model used to compare convergence true behavior begins to become muddled with computational artifacts. Any solution compared to another sufficiently close solution will display erroneously high convergence rates, assuming the solution converges in that direction. The calculated convergence rate of 2.0599 agrees well with the expected rate of 2 based on the order of the truncation error of the finite differences. In order to more accurately assess the convergence rate the code was optimized until the limiting factor was memory, permitting construction of a 2049x2049 node solution for comparison.



Figure 6: Convergence of finite difference solution wrt. 2049x2049 node solution.

*6. Compute the maximum deflection in the membrane for the following ranges of angles,* θ*= 0-40° and,* φ *= 0-40°, for an increment of 2°. Plot the result using the Matlab function contour.*

Figure 7 is a plot of the dependence of the maximum deflection on the physical domain angles. A surface plot was generated in place of a contour plot as more information was able to be seen with the interpolation. As expected as the membrane grew narrower the maximum deflection decreased. This is to be expected; the closer proximity of the boundary conditions limits local deformation. Complementarily the greatest deflection was seen when the membrane was square as this configuration has the BCs located furthest away from the centroid.

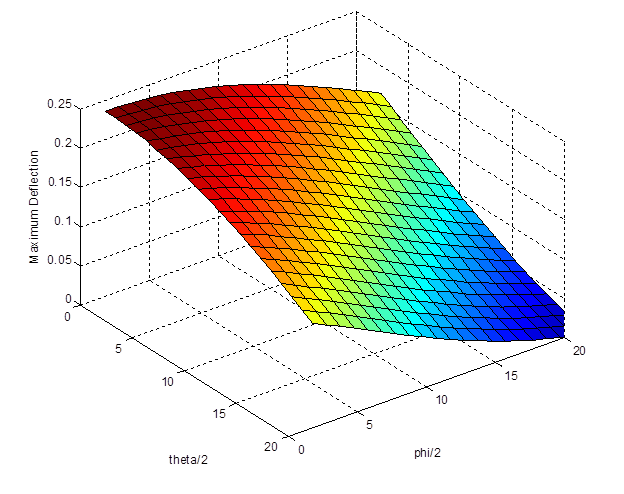


Figure 7: Dependence of maximum deflection on shape of physical domain.

**Additional Comment: Execution speed improvements**

For large matrices conventional matrices demand far too much memory for the little amount of data they encode for problems such as this with relatively few non-zero diagonals. The skeleton code provided with the assignment makes use of spalloc() to pre-allocate a sparse matrix and then puts stencil values in by directly indexing desired positions.

While this is simple to implement it presents significant processing overhead as sparse matrices are not optimized for access in this way. An alternate method of construction of the ‘A’ matrix was devised making use of spdiags(). This function takes as input two matrices, one containing element values and implicit row positions that map 1:1 to the sparse matrix row position, the other containing explicit column locations in the sparse matrix for each of the columns in the value matrix. The ‘A’ matrix diagonals are pre-calculated, then the sparse matrix is created from the pre-calculated values.

This method is far more efficient CPU time-wise and offers significant execution speeds that continue to grow as the sparse matrix increases in size. Figure 8 offers measurements of execution times for each method. For a qualitative comparison N=513 took the spalloc() method 200 seconds while only taking the spdiag() method 2 seconds.

Figure 8: Significant gains achieved by using spdiags()

**Appendix A – Program Code**

clear all

close all

tic

FinalOrder=5; %(2^FinalOrder)+1 number of nodes used along one side

RunMode = 1; %1 for normal operation, 2 to examine error, 3 for execution testing

MapType = 1; %1 for constant transform, 2 for nodal evaluation of transform

SparseType = 2; %1 for indexed population, 2 for apriori creation

if MapType==2 && SparseType==2

error('Unsupported mode types')

end

%Non-loop required initialization

%=========================================================

%Comp domain params

% xi-eta, xi, eta, offset

CompDomain = [0 0 0 1; %a

0 1 0 1; %b

1 1 1 1; %c

0 0 1 1]; %d

% Physical domain parameters

L = 2;

theta = 0;

phi = 0;

Forcing = 1; % Forcing function

%BCs

Left =0;

Right =0;

Top =0;

Bottom =0;

% a, b , c , d

PhysDomainX = [0, L\*cos(theta), (L\*cos(theta))+(L\*sin(phi)), L\*sin(phi)];

PhysDomainY = [0, L\*sin(theta), (L\*sin(theta))+(L\*cos(phi)), L\*cos(phi)];

% Calculate bilinear map coefficients

BiMapX = CompDomain\PhysDomainX';

BiMapY = CompDomain\PhysDomainY';

for Exp=FinalOrder:-1:1

N=(2^Exp)+1 %Left without semicolon to indicate progress

% Comp domain params

%=========================================================

Nxi = N;

Neta = N;

NNodes = Nxi\*Neta;

F = ones(1,NNodes);

Dxi = 1/(Nxi - 1);

Deta = 1/(Neta - 1);

% Calculate xi, eta and node number

%=========================================================

if SparseType==1

NodeNumber=zeros(Neta,Nxi);

counter = 0;

for j = 1:Neta

for i = 1:Nxi

counter = counter + 1;

NodeNumber(i,j) = counter;

end

end

end

% Implement the mapping

%=========================================================

%It turns out that the transform coefficients are constant across (i,j) for

%this mapping so we can save a lot of time not calculating them for all

%i,j) or indexing the matrices in later equations.

% Calculate the transformed equation coefficients

if MapType==1

a= BiMapX(3)^2+BiMapY(3)^2;

b= BiMapX(3)\*BiMapX(2)+BiMapY(3)\*BiMapY(2);

c= BiMapX(2)^2+BiMapY(2)^2;

alpha = -2\*b\*BiMapX(1);

beta = -2\*b\*BiMapY(1);

J= BiMapX(2)\*BiMapY(3)-BiMapX(3)\*BiMapY(2);

d= (BiMapY(2)\*alpha-BiMapX(2)\*beta)/J;

e= BiMapX(3)\*beta-BiMapY(3)\*alpha;

elseif MapType==2

for j=1:Neta

for i=1:Nxi

xi=(i-1)\*Neta\*Dxi;

eta=(j-1)\*Nxi\*Deta;

a(i,j)= ((BiMapX(1)\*xi+BiMapX(3))^2)+((BiMapY(1)\*xi+BiMapY(3))^2);

b(i,j)= ((BiMapX(1)\*xi+BiMapX(3))\*(BiMapX(1)\*eta+BiMapX(2)))+((BiMapY(1)\*xi+BiMapY(3))\*(BiMapY(1)\*eta+BiMapY(2)));

c(i,j)= ((BiMapX(1)\*eta+BiMapX(2))^2)+((BiMapY(1)\*eta+BiMapY(2))^2);

alpha = -2\*b(i,j)\*BiMapX(1);

beta = -2\*b(i,j)\*BiMapY(1);

J(i,j)= ((BiMapX(1)\*eta+BiMapX(2))\*(BiMapY(1)\*xi+BiMapY(3)))-((BiMapX(1)\*xi+BiMapX(3))\*(BiMapY(1)\*eta+BiMapY(2)));

d(i,j)= (((BiMapY(1)\*eta+BiMapY(2))\*alpha)-((BiMapX(1)\*eta+BiMapX(2))\*beta))/J(i,j);

e(i,j)= ((BiMapX(1)\*xi+BiMapX(3))\*beta)-((BiMapY(1)\*xi+BiMapY(3))\*alpha);

end

end

end

% Set-up finite difference matrix

%=========================================================

if MapType==1

corner =(2\*b)/(4\*Deta\*Dxi)\*(-1/J^2);

ip =(a/Dxi^2 +e/(2\*Dxi))\*(-1/J^2);

in =(a/Dxi^2 -e/(2\*Dxi))\*(-1/J^2);

jp =(c/Deta^2 +d/(2\*Deta))\*(-1/J^2);

jn =(c/Deta^2 -d/(2\*Deta))\*(-1/J^2);

center =((-2\*c)/Deta^2 -(2\*a)/Dxi^2)\*(-1/J^2);

if SparseType==1

A = spalloc(NNodes,NNodes, NNodes\*10); % allocate the matrix

for j = 2:(Neta - 1)

for i = 2:(Nxi-1)

% Populate A-Matrix

%================================================

NodeN = NodeNumber(i,j);

A(NodeN, NodeNumber(i+0,j+0)) = center;

A(NodeN, NodeNumber(i+1,j+0)) = ip;

A(NodeN, NodeNumber(i-1,j+0)) = in;

A(NodeN, NodeNumber(i+0,j+1)) = jp;

A(NodeN, NodeNumber(i+0,j-1)) = jn;

A(NodeN, NodeNumber(i+1,j+1)) = -corner;

A(NodeN, NodeNumber(i+1,j-1)) = corner;

A(NodeN, NodeNumber(i-1,j+1)) = corner;

A(NodeN, NodeNumber(i-1,j-1)) = -corner;

end

end

elseif SparseType==2

%Create matrix containing diagonals to be put in A matrix

%Brother are directly adjacent diags(nodes along i in stencil), cousins are the

%grouped diags(each level along j in stencil)

%"Magic" numbers are static and based on a 3x3 stencil

DiagValue = ones(NNodes,9);

for cousin =0:2

for brother =0:2

for iter =1:Neta-3

R =Nxi\*iter-1:Nxi\*iter;

%Snips out gaps amongst diags in A

DiagValue(R+brother+cousin\*Nxi,cousin\*3+brother+1) =0;

end

%Cuts off top and bottom of A

DiagValue([1:brother+cousin\*Nxi, NNodes-(2-cousin)\*Nxi+brother-1:NNodes],cousin\*3+brother+1) =0;

end

end

%Set diags to correct static value

DiagValue(:,1) =DiagValue(:,1).\*-corner;

DiagValue(:,2) =DiagValue(:,2).\*jn;

DiagValue(:,3) =DiagValue(:,3).\*corner;

DiagValue(:,4) =DiagValue(:,4).\*in;

DiagValue(:,5) =DiagValue(:,5).\*center;

DiagValue(:,6) =DiagValue(:,6).\*ip;

DiagValue(:,7) =DiagValue(:,7).\*corner;

DiagValue(:,8) =DiagValue(:,8).\*jp;

DiagValue(:,9) =DiagValue(:,9).\*-corner;

%Set-up BCs

%BC finite differences

DiagValue(1:Nxi,5) = 1; %Bottom

DiagValue(NNodes-Nxi+1:NNodes,5) = 1; %Top

DiagValue(Nxi+1:Nxi:NNodes-Nxi+1,5) = 1; %Left

DiagValue(2\*Nxi:Nxi:NNodes-Nxi,5) = 1; %Right

%BC values

F(1:Nxi)=Bottom;

F(NNodes-Nxi+1:NNodes) =Top;

F(Nxi+1:Nxi:NNodes-Nxi+1) =Left;

F(2\*Nxi:Nxi:NNodes-Nxi) =Right;

%Create sparse matrix with diags defined in DiagValue along diag numbers

%defined in DiagLoc

%Based on 3x3 stencil

DiagLoc =[-Nxi-1 -Nxi -Nxi+1 -1 0 1 Nxi-1 Nxi Nxi+1];

A=spdiags(DiagValue, DiagLoc, NNodes, NNodes);

end

elseif MapType==2

A = spalloc(NNodes,NNodes, NNodes\*10); % allocate the matrix

for j = 2:(Neta - 1)

for i = 2:(Nxi-1)

% Populate A-Matrix

%================================================

NodeN = NodeNumber(i,j);

A(NodeN, NodeNumber(i+0,j+0)) = ((-2\*c(i,j))/Deta^2 -(2\*a(i,j))/Dxi^2) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i+1,j+0)) = (a(i,j)/Dxi^2 +e(i,j)/(2\*Dxi)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i-1,j+0)) = (a(i,j)/Dxi^2 -e(i,j)/(2\*Dxi)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i+0,j+1)) = (c(i,j)/Deta^2 +d(i,j)/(2\*Deta)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i+0,j-1)) = (c(i,j)/Deta^2 -d(i,j)/(2\*Deta)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i+1,j+1)) = ((-2\*b(i,j))/(4\*Deta\*Dxi)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i+1,j-1)) = ((2\*b(i,j))/(4\*Deta\*Dxi)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i-1,j+1)) = ((2\*b(i,j))/(4\*Deta\*Dxi)) \*(-1./J(i,j)^2);

A(NodeN, NodeNumber(i-1,j-1)) = ((-2\*b(i,j))/(4\*Deta\*Dxi)) \*(-1./J(i,j)^2);

end

end

end

%Populate forcing vector

%================================================

F=F.\*Forcing;

% Implement Boundary conditions

%=========================================================

if SparseType==1

for(i = 1:(Nxi))

j = 1;

ij = NodeNumber(i,j);

A(ij,ij) = Bottom;

F(ij) = 0;

j = Neta;

ij = NodeNumber(i,j);

A(ij,ij) = Top;

F(ij) = 0;

end

for(j = 2:(Neta-1))

i = 1;

ij = NodeNumber(i,j);

A(ij,ij) = Left;

F(ij) = 0;

i = Nxi;

ij = NodeNumber(i,j);

A(ij,ij) = Right;

F(ij) = 0;

end

end

% Solve the linear system

%=========================================================

Solution = A\F';

MaxDeflect = max(max(Solution));

if Exp==FinalOrder

Exact=MaxDeflect;

else

InfNormError(Exp)=abs(Exact-MaxDeflect);

end

Saved(Exp)=MaxDeflect; %Look for spurious behavior, this should stay mostly constant

if RunMode~=2 %Break after 1 loop for normal mode

break

end

end

%Plot solved function for a particular N

if RunMode==1

Deflection = reshape(Solution, Nxi, Neta);

figure

surf(0:Dxi\*Neta:Nxi,0:Deta\*Neta:Neta,Deflection)

colorbar

axis equal

view([0 0 1])

title('Computational Domain')

%Find mapped locations to plot physical domain

for j=1:Neta

for i=1:Nxi

xi=(i-1)\*Dxi;

eta=(j-1)\*Deta;

X(i,j) = (BiMapX(1)\*(xi\*eta))+ (BiMapX(2)\*xi) + (BiMapX(3)\*eta) + BiMapX(4);

Y(i,j) = (BiMapY(1)\*(xi\*eta))+ (BiMapY(2)\*xi) + (BiMapY(3)\*eta) + BiMapY(4);

end

end

figure

surf(X,Y,Deflection)

colorbar

axis equal

view([0 0 1])

title('Physical Domain')

end

%Plot error, find slope of loglog curve

if RunMode==2

N = 1:FinalOrder-1; %Exclude final order itself

N = (2.^N)+1;

loglog(N',InfNormError, '-r')

loglogfit = polyfit(log(N),log(InfNormError),1);

text(N(floor(FinalOrder/2)),InfNormError(floor(FinalOrder/2)),num2str(loglogfit(1)))

end

toc

beep