```
!=========================================================================!
!                              main program                               !
!=========================================================================!
program FEM2D
use quadrature
use ParamModule
use MumpsModule
implicit none
 integer :: i,k
 double precision :: T0,T2,T3,T4
 Type (Param),target::theParam
 Type (AIJ),target:: sparse
 Type (quad),target::qd
 Type (BasFunc),target :: bf
call Initialization (TheParam,"param.dat")
    subroutine Initialization (TheParam,filename)                    !
        !-----------------------------------------------------------------------------!
        use ParamModule
        use quadrature
        implicit none
        Character*(*), INTENT (IN)  :: filename
        Integer :: datafile = 1
        Type (Param),target::TheParam
        Type (AIJ),target:: sparse
        Type (quad),target::qd
        Type (BasFunc),target :: bf
        open (datafile,file=filename)
        Call Init (TheParam,datafile,filename)
            subroutine Init (prm,datafile,filename)
                Character*(*), INTENT (IN)  :: filename
                Type(Param), INTENT (INOUT) :: prm
                Integer :: datafile ! = 1
                Integer :: i,j,k,m,icounter
                ! domain size
                read(datafile,*)
                read(datafile,*)
                read(datafile,*)
                :
                read(datafile,*) prm%xmin
                read(datafile,*) prm%xmax
                read(datafile,*) prm%ymin
                read(datafile,*) prm%ymax

                ! number of numerical constants
                read(datafile,*)
                read(datafile,*)
                read(datafile,*)
                read(datafile,*) prm%nbNC
                allocate (prm%NumCst (prm%nbNC))

                do i=1,prm%nbNC
                read(datafile,*) prm%NumCst (i)
                end do

                read(datafile,*) !-------------------------!
                read(datafile,*) !   physical constants    !
                read(datafile,*) !-------------------------!
                read(datafile,*) prm%nbPC
                allocate (prm%PhysCst (prm%nbPC))
                do i=1,prm%nbPC
                read(datafile,*) prm%PhysCst (i)
                end do

                prm%h(1) = (prm%xmax-prm%xmin) / dble(prm%numCst(2)-1)
                prm%h(2) = (prm%ymax-prm%ymin) / dble(prm%numCst(3)-1)
                !penalization coefficient
                prm%delta = prm%PhysCst(6) * min (prm%h(1),prm%h(2))
                read(datafile,'(A)') prm%bctop
                read(datafile,'(A)') prm%bcbottom
                read(datafile,'(A)') prm%bcleft
                read(datafile,'(A)') prm%bcright
                !x and y coordinates
                allocate (prm%x(prm%numCst(2)),prm%y(prm%numCst(3)))
```

imports all modules

i and k are integers

datafile=1

this line opens the datafile

reads it

line1 -> line2 -> 3 -> :

each line reads a line from the file

prm is a Param Object

those basically say 'this line, -1, is the Xmin value of the prm object of type Param'

number of parameters, in the datafile

runs its points in x,y, etc

for loop for all Parameters

Same thing really

looks like h(1) is dx, h(2) = dy

so has just using dx ordy i guess.

boundary conditions

generate

```fortran
prm%x(1) = prm%xmin
prm%y(1) = prm%ymin
do i = 2,prm%NumCst(2)
prm%x(i) = prm%x(i-1) + prm%h(1)
end do
do i = 2,prm%NumCst(3)
prm%y(i) = prm%y(i-1) + prm%h(2)
!write (*,*) prm%y(i)
end do
```

x and y array

fills them in with coordinates, for loop style

```fortran
!mount/allocate local element coordinates
prm%neX = prm%NumCst(2) - 1
prm%neY = prm%NumCst(3) - 1
prm%Tne = prm%neX*prm%neY
prm%Tnp = prm%NumCst(2)*prm%NumCst(3)
!xg and yg coordinates
allocate (prm%xg(prm%NumCst(2)*prm%NumCst(3)),prm%yg(prm%NumCst(2)*prm%NumCst(3)))
m = 1
do i = 1,prm%NumCst(2)*prm%NumCst(3)
prm%xg(i) = prm%x(m)
m = m + 1
if (m .gt. prm%NumCst(2)) m = 1
end do
m = 1
do i = 1,prm%NumCst(2)*prm%NumCst(3)
prm%yg(i) = prm%y(m)
if (i .eq. m*prm%NumCst(2)) m = m + 1
end do
```

(no. of Elements) — total no. of edges — Total no. of points

→ generates a 2d grid from x and y for some reason

its like a matlab xg and yg matrix. It seems like → (if m > a number)

```fortran
allocate(prm%leX(prm%neX*prm%neY,prm%numCst(1)))
allocate(prm%leY(prm%neX*prm%neY,prm%numCst(1)))

icounter = 1
do i = 1,prm%Tne
prm%leX(i,1) = prm%x(icounter)
prm%leX(i,2) = prm%x(icounter) + prm%h(1)
prm%leX(i,3) = prm%x(icounter) + prm%h(1)
prm%leX(i,4) = prm%x(icounter)
icounter = icounter + 1
if (icounter .gt. prm%neX) icounter = 1
end do
icounter = 1
j = 1
k = 1
do i = 1,prm%Tne
prm%leY(i,1) = prm%y(j)
prm%leY(i,2) = prm%y(j)
prm%leY(i,3) = prm%y(j) + prm%h(2)
prm%leY(i,4) = prm%y(j) + prm%h(2)
icounter = icounter + 1
if (icounter .gt. k*prm%neX) then
    j = j + 1
    k = k + 1
end if
end do
```

← for all Elements

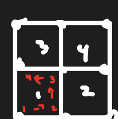
whats this again? — make edge thing? — for all x

$$\begin{vmatrix} -1 & -0.7 & -0.7 & -1 \\ -0.7 & -0.5 & -0.5 & -0.7 \end{vmatrix}$$

ah so until it gets to, it loops over all x in a single y, then moves onto the next y, and so on

prm%leXY (k, n), returns x and y coordinates (k → element number, n → node number)

so its actually, but in my examples its tentative, oh well




```fortran
print *,'---------------------'
print *,'ParamInitialize : done'
print *,'---------------------'
end subroutine Init
call LGM (TheParam)
  subroutine LGM (prm)
implicit none
Type(Param), INTENT (INOUT) :: prm
integer :: k,m
! ========================== !
!         Map_loc            !
! ========================== !
allocate (prm%Lgm(prm%neX*prm%neY,prm%numCst(1)))
prm%Lgm(:,:) = 0
m = 1
do k = 1, prm%neX*prm%neY
if (k .eq. m*prm%neX) then
    prm%Lgm(k,1) = k + (k/prm%neX) - 1
    m = m + 1
else
    prm%Lgm(k,1) = k + (k/prm%neX)
end if
prm%Lgm(k,2) = prm%Lgm(k,1) + 1
prm%Lgm(k,3) = prm%Lgm(k,2) + prm%NumCst(2)
```

initializes the grid and environmental parameters finished

do for all elements (1,2,3,4)

okay so LGM Returns the vertex number of the elements.

Lgm(k, m) = n (k → element, m → local node, n → global node)

it $\frac{1}{2}$ (but int)

so for element 1, its (1, 2, 5, 4)
element 2, its (2, 3, 6, 5)

```
            prm%Lgm(k,4) = prm%Lgm(k,1) + prm%NumCst(2)
            end do
            !
            end subroutine LGM
```

Ask what NBE and LGM are

```
        call NBE (TheParam)
            subroutine NBE (prm)
                implicit none
                Type(Param), INTENT (INOUT) :: prm
                integer :: k
                double precision :: left,right,bottom,top
                allocate (prm%Nbe(prm%neX*prm%neY,8))
                prm%Nbe(:,:) = 0
                left   = prm%xmin
                right  = prm%xmax
                bottom = prm%ymin
                top    = prm%ymax
                do k = 1, prm%neX*prm%neY
                    ! Neighbor 1
                    if (prm%leX(k,1).gt.left .AND. prm%leY(k,1).gt.bottom) then
                    prm%Nbe(k,1) = k - (prm%neX + 1)
                    end if
                    ! Neighbor 2
                    if (prm%leY(k,1).gt.bottom) then
                    prm%Nbe(k,2) = k - prm%neX
                    end if
                    ! Neighbor 3
                    if (prm%leX(k,2).lt.right .AND. prm%leY(k,2).gt.bottom) then
                    prm%Nbe(k,3) = k - (prm%neX - 1)
                    end if
                    ! Neighbor 4
                    if (prm%leX(k,2).lt.right) then
                    prm%Nbe(k,4) = k + 1
                    end if
                    ! Neighbor 5
                    if (prm%leY(k,3).lt.top .AND. prm%leX(k,3).lt.right) then
                    prm%Nbe(k,5) = k + (prm%neX + 1)
                    end if
                    ! Neighbor 6
                    if (prm%leY(k,3).lt.top) then
                    prm%Nbe(k,6) = k + prm%neX
                    end if
                    ! Neighbor 7
                    if (prm%leY(k,4).lt.top .AND. prm%leX(k,4).gt.left) then
                    prm%Nbe(k,7) = k + (prm%neX - 1)
                    end if
                    ! Neighbor 8
                    if (prm%leX(k,1).gt.left) then
                    prm%Nbe(k,8) = k - 1
                    end if
                end do
                end subroutine NBE
        end subroutine Initialization
```

-> initializes NBE matrix

Does NBE mean neighboring elements?
yes!

done for all elements

neighbors

1 2 3
8 X 4
7 6 5

So basically, it returns an array of what element is. eg,

Prm%NBE(3,4) returns the right neighbor of element 3

-> findow what properties qd has

```
call Matrix_A (TheParam,qd,bf,sparse)
    subroutine Matrix_A (TheParam,qd,bf,sparse)                                    !
        !-------------------------------------------------------------------------------!
        use ParamModule
        use quadrature
        implicit none
        Type (Param),target::theParam
        Type (AIJ),target:: sparse
        Type (quad),target::qd
        Type (BasFunc),target :: bf
        call quad_calc (TheParam,qd)        ! Calling abscissas and weightings
            subroutine quad_calc(par,qd)
                use ParamModule
                implicit none
                type (Param) :: par
                type (quad) :: qd
                integer :: i,j
                allocate (qd%quad_x0(par%NumCst(4)),qd%quad_w(par%NumCst(4)))
                ! initialization
                qd%quad_w  = 0D0
                qd%quad_x0 = 0D0
                select case (par%NumCst(4))
                case (1)
                qd%quad_w (1) = 1D0
                case (2)
                qd%quad_w (1) = 1D0
```

It works like it's using quadratic instead of linear?

Par = prm

number of quadrature points

w = weights
x = no of points

```
            qd%quad_w (2) = 1D0
        !
            qd%quad_x0 (1) = sqrt (1D0/3D0)
            qd%quad_x0 (2) = -sqrt (1D0/3D0)
            case (3)
            qd%quad_w (1) = 0.55555555555555555555555555555556D0
            qd%quad_w (2) = 0.88888888888888888888888888888889D0
            qd%quad_w (3) = 0.55555555555555555555555555555556D0
        !
            qd%quad_x0 (1) =  0.77459666924148337703585307995 6D0
            qd%quad_x0 (3) = -0.77459666924148337703585307995 6D0
            case (4)
            qd%quad_w (1) = 0.34785484513745385737306394922 2D0
            qd%quad_w (2) = 0.65214515486254614262693605077 8D0
            qd%quad_w (3) = 0.65214515486254614262693605077 8D0
            qd%quad_w (4) = 0.34785484513745385737306394922 2D0
        !
            qd%quad_x0 (1) =  0.86113631159405257522394648889 3D0
            qd%quad_x0 (2) =  0.33998104358485626480266575910 3D0
            qd%quad_x0 (3) = -0.33998104358485626480266575910 3D0
            qd%quad_x0 (4) = -0.86113631159405257522394648889 3D0
            case (5)
            qd%quad_w (1) = 0.23692688505618908751426404072 0D0
            qd%quad_w (2) = 0.47862867049936646804129151483 6D0
            qd%quad_w (3) = 0.56888888888888888888888888888 9D0
            qd%quad_w (4) = 0.47862867049936646804129151483 6D0
            qd%quad_w (5) = 0.23692688505618908751426404072 0D0
        !
            qd%quad_x0 (1) =  0.90617984593866399279762687829 9D0
            qd%quad_x0 (2) =  0.53846931010568309103631442070 0D0
            qd%quad_x0 (4) = -0.53846931010568309103631442070 0D0
            qd%quad_x0 (5) = -0.90617984593866399279762687829 9D0
            case (7)
            qd%quad_w (1) = 0.12948496616887 0D0
            qd%quad_w (2) = 0.27970539148927 7D0
            qd%quad_w (3) = 0.38183005050511 9D0
            qd%quad_w (4) = 0.41795918367346 9D0
            qd%quad_w (5) = 0.38183005050511 9D0
            qd%quad_w (6) = 0.27970539148927 7D0
            qd%quad_w (7) = 0.12948496616887 0D0
        !
            qd%quad_x0 (1) =  0.94910791234275 9D0
            qd%quad_x0 (2) =  0.74153118559939 4D0
            qd%quad_x0 (3) =  0.40584515137739 7D0
            qd%quad_x0 (5) = -0.40584515137739 7D0
            qd%quad_x0 (6) = -0.74153118559939 4D0
            qd%quad_x0 (7) = -0.94910791234275 9D0
            end select
        end subroutine quad_calc
call calAloc (TheParam,qd,bf)       ! Calculating local mat. entries for every element
    subroutine calAloc(par,qd,bf)
        !
        use ParamModule
        use quadrature
        implicit none
        type (Param) :: par
        type (quad) :: qd
        type (BasFunc) :: bf
        !
        double precision :: F_xy,dx,dy,xr,yr,sr,pi,kxy,xmin,xmax,ymin,ymax,xe,ye
        double precision :: sigma
        integer :: i,j,k,m,n,l,nobs
        double precision, dimension (:),allocatable :: a,b,c,d
        double precision :: eps,diam,conv,diff,wx,wy
        !


        diam = par%PhysCst(7)
        nobs = par%numCst(6)+1
        allocate (bf%Aloc(par%Tne,par%NumCst(1),par%NumCst(1)))
        allocate (bf%f(par%NumCst(1)),bf%dxf(par%NumCst(1)),bf%dyf(par%NumCst(1)))

        allocate (a(nobs),b(nobs),c(nobs),d(nobs))

        eps =( par%xmax - par%xmin) / dble(nobs-1)
        if (nobs .eq. 0) eps = 0D0
        a(:) = 10D0
        do m=1,nobs
```

So basically, using this $\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$

(2) = 0

This is from gaussian quadrature

Okay right?

important!

Base function?

diameter of obstacles

amount of obstacles in one axis (?)

Just creating the arrays

Probably epsilon ε for penalization

eps = (xmax - xmin) / no. of obstacles

not sure what eps stands for tho

why would you get amount of obstacles to be -1?

```
        a(m) = par%xmin + m*eps - eps!0.5*eps
        end do
        b = a + 0.5D0*diam*eps
        a = a - 0.5D0*diam*eps
        c = a
        d = b
        do k = 1,par%Tne
        xmin = par%lex(k,1)
        xmax = par%lex(k,2)
        ymin = par%ley(k,1)
        ymax = par%ley(k,4)
        xe = (xmin + xmax) / 2D0
        ye = (ymin + ymax) / 2D0
        dx = xmax - xmin
        dy = ymax - ymin
        wx = 2D0 * ye * ( 1 - xe**2 )!0D0!
        wy = -2D0 * xe * ( 1 - ye**2 ) !1D0!
        sigma = 0D0
        do m=1,nobs
        do n=1,nobs
            if (xe.ge.a(m) .AND. xe.le.b(m) .AND. ye.ge.c(n) .AND. ye.le.d(n)) then
                sigma = 1D0 / ( par%delta**3 )
            end if
        end do
        end do
        do m = 1,par%NumCst(1)
        do n = 1,par%NumCst(1)
        !
            bf%Aloc(k,m,n) = 0D0
            do i = 1,par%NumCst(4)
            do j = 1,par%NumCst(4)
        !
                xr = (dx/2.)*qd%quad_x0(i) + (dx/2.)
                yr = (dy/2.)*qd%quad_x0(j) + (dy/2.)
        !       basis functions
                bf%f(1) = (1. - xr/dx) * (1. - yr/dy)
                bf%f(2) = (xr/dx) * (1. - yr/dy)
                bf%f(3) = (xr/dx) * (yr/dy)
                bf%f(4) = (1. - xr/dx) * (yr/dy)
        ! basis functions derivatives
                bf%dxf(1) = (-1./dx) + (yr/(dx*dy))
                bf%dyf(1) = (-1./dy) + (xr/(dx*dy))
                bf%dxf(2) = (1./dx) - (yr/(dx*dy))
                bf%dyf(2) = (-xr/(dx*dy))
                bf%dxf(3) = (yr/(dx*dy))
                bf%dyf(3) = (xr/(dx*dy))
                bf%dxf(4) = (-yr/(dx*dy))
                bf%dyf(4) = (1./dy) - (xr/(dx*dy))
        ! Oscillating functions k(x,y) = sin(2*pi*x)sin(2*pi*y)
        !            kxy = sin(2*pi*xr)*sin(2*pi*yr)
        !
        conv = ( wx * bf%dxf(m) + wy * bf%dyf(m) ) * bf%f(n)
        diff = par%PhysCst(8)*( bf%dxf(m)*bf%dxf(n) + bf%dyf(m)*bf%dyf(m) )
        F_xy =  diff + conv  + (sigma*bf%f(m)*bf%f(n))
        bf%Aloc(k,m,n)=bf%Aloc(k,m,n)+qd%quad_w(i)*qd%quad_w(j)*F_xy*((dx*dy)/4.)
        !
                end do
            end do
        !
            end do
        end do
        end do
        !
        end subroutine calAloc
    call GlobalMap (TheParam,sparse)    ! Constructing sparse matrix mapping mechanism
        subroutine GlobalMap (par,sparse)
            use ParamModule
            implicit none
            integer :: k,n,m,k1,k2,n1,n2,m1
            type (Param) :: par
            type (AIJ) :: sparse
            sparse%nonzero = 0
            allocate (sparse%GML(par%Tne,par%NumCst(1),par%NumCst(1)))
            sparse%GML(:,:,:) = 0
            do k = 1, par%Tne
            do n = 1, par%NumCst(1)
                do m = 1, par%NumCst(1)
                if (sparse%GML(k,n,m) == 0) then
                sparse%nonzero = sparse%nonzero + 1
```

a = lower x
b = higher x
c = lower y
d = higher y

:,:,1 Points of the solid wedge
Coordinates of the points making up the obstacle

Velocity vector w

Should ask what eps is

Ymin not -inf

for 3 points in x and y

so it loops 4 times in this case

midpoint of the two nodes, then element coordinates

constant edge

$2.0.5.(1-0.5^2)$

Obstacle editing

do for all points in element

Get logic to make way through this subroutine

for all elements: Calculate parameters, and w?

for all nodes: / for all nodes: } get the 4x4 matrix

! Goal is to calculate
! $\int \nabla \cdot \nabla$ for diffusion.
! but it is easier to use
! $\Sigma \nabla \cdot \nabla$ instead, at specific points
! using quadrature
Set value = 0 first.
for all Quadrature points in a node:
Calculate
$N_1, N_2, \nabla N_1, \nabla N_2$ values
at all quadrature points

for $\nabla N_1 \cdot \nabla N_2$ lets say calculates values of ∇N_1 and ∇N_2 here.

well the code actually calculates all N_1, N_2, N_3, N_4 and then picks the relevant ones. Oh it could be more efficient

convection term

Diffusion term

Penalisation

If we triangle as frame, so maybe also for this rectangular thing

$N_1 = (1-\frac{x}{dx})(1-\frac{y}{dy}) = (-\frac{1}{dx})(1-\frac{y}{dy}) + 0$ ✓

$\nabla N_1 = (-\frac{1}{dx}) + \frac{y}{dx\,dy} = -\frac{1}{dx} + \frac{y}{dx\,dy}$ ✓

So far it looks like its all calculated in the reference element. Is he going to move it into the global thing after?

I get it!!!

Pretty sure its converting the local matrix into the global matrix
Also its going to use a sparse matrix

∴ in our simplified scenario, gives a 4x4x4 matrix

no. of elements
no. of nodes
no. of nodes

for all elements
for all node pairs

for loop goes over all this

If element is not 0, skip all this.

If 0, then nonzero +1

```
                sparse%GML(k,n,m) = sparse%nonzero
                do k1 = 1, 8                                  ! loops Ground neighboring elements of k
                    k2 = par%Nbe(k,k1)
                    if (k2 /= 0) then                         ! if a neighbor exists
                    n1 = 0
                    m1 = 0
                    do n2 = 1, par%NumCst(1)                  ! forall nodes in that element
                        if (par%Lgm(k,n) == par%Lgm(k2, n2)) then
                        n1 = n2                               ! Current n node = neighboring node
                        end if
                        if (par%Lgm(k,m) == par%Lgm(k2, n2)) then
                        m1 = n2                               ! Current m node = neighboring node.
                        end if
                    end do
                    if ((n1 /= 0) .AND. (m1 /= 0)) then       ! if neighboring node exists
                        sparse%GML(k2,n1,m1) = sparse%nonzero
                    end if                                    ! neighbor element  ! at these nodes  ! its a counter
                end if
                end do
                end do
            end if
        end do
    end do
    end do
end subroutine GlobalMap                                      ! Its used for the ∫∇φi·∇φj calc my guess.
                                                              ! So Global Map returns a map of all things which are adjacent to each other
    call bcond (TheParam,sparse)        ! Applying B.C. (calculate ~%Nbc)
        subroutine bcond (prm,sp)   ! Reading B.C. values for Lag. multipliers input    !
            ! ================================================================================ !
            use ParamModule
            implicit none
            type (Param)    :: prm
            type (AIJ)      :: sp
            !
            integer :: i,j,k,l,m,lt,tn
            character :: d,n
            allocate (prm%tnode(prm%NumCst(2)),prm%bnode(prm%NumCst(2)))   ! top  bottom
            allocate (prm%lnode(prm%NumCst(3)),prm%rnode(prm%NumCst(3)))   ! left  right
            !                                                              ! prm%node is a number array with as many numbers as Points m x, y
            j = 0
            if (prm%bctop .eq. 'd')    j = j + prm%NumCst(2)               ! from datfile
            if (prm%bcleft .eq. 'd')   j = j + prm%NumCst(3)               ! j is no. of boundary nodes (overestimated)
            if (prm%bcright .eq. 'd')  j = j + prm%NumCst(3)
            if (prm%bcbottom .eq. 'd') j = j + prm%NumCst(2)
            prm%Nbc = j !amount  of boundary conditioned nodes grossly estimated
            allocate (prm%qbc(prm%Nbc),prm%qbcval(prm%Nbc))
            ! nodes of boundary                                            ! not sure what qbc and qbcval are yet
            tn  = prm%Tnp                                                  ! total number of nodes
            lt  = tn - prm%neX                                             ! looks like a temporary counter
            prm%tnode(:) = 0
            prm%bnode(:) = 0
            prm%lnode(:) = 0                                               ! cleans up the nodes
            prm%rnode(:) = 0
            prm%qbc(:)   = 0
            prm%qbcval(:)= 0D0
            ! ==================== top  ======================== !
            j = 0                                                          ! resets j
            do i = 1,prm%NumCst(2)                                         ! for all top boundary conditions (the bnodes now simple example)
                prm%tnode(i) = lt + (i-1)
            end do                                                         ! lists all the top boundary nodes (eg. 7,8,9)
            if (prm%bctop .eq. 'd') then
            do i = 1,prm%NumCst(2)                                         ! for all boundary nodes on top
                prm%qbc(i) = prm%tnode(i)                                  ! sets the q node
                prm%qbcval(i) = prm%PhysCst(1)                             ! sets the value at that node (usually 0)
                !prm%qbcval(i) = 0D0                                       ! QNODE 1
                !prm%qbcval(i) = 0D0                                       ! QNODE 2
                !prm%qbcval(i) = ((prm%x(i)-prm%xmin)/(prm%xmax-prm%xmin))          ! QNODE 3
                !prm%qbcval(i) = (1D0 - ((prm%x(i)-prm%xmin)/(prm%xmax-prm%xmin)))  ! QNODE 4
                !prm%qbcval(i) = sin (3*3.14159265359*prm%xg(i)/prm%xmax) ! FOR CONVERGENCE CHECK
            end do
            j = j + prm%NumCst(2)                                          ! now sets j = 3, finishes the top boundary conditions
            end if
            ! ==================== bottom ======================== !
            do i = 1,prm%NumCst(2)                                         ! for all edge nodes
                prm%bnode(i) = i                                           ! list of bottom nodes
            end do
            if (prm%bcbottom .eq. 'd') then
            do i = 1,prm%NumCst(2)                                         ! (1,2,3)
                prm%qbc(j+i) = prm%bnode(i)                                ! 4,5,6.
                !prm%qbcval(j+i) = (1D0 -((prm%x(i)-prm%xmin)/(prm%xmax-prm%xmin))) ! QNODE 1
                !prm%qbcval(j+i) = ((prm%x(i)-prm%xmin)/(prm%xmax-prm%xmin))        ! QNODE 2
                !prm%qbcval(j+i) = 0D0                                              ! QNODE 3
```

```
            prm%qbcval(j+i) = prm%PhysCst(2)
            !prm%qbcval(j+i) = 0D0                                    ! QNODE 4
        end do
        j = j + prm%NumCst(2)
        end if
        do i = 1,prm%NumCst(3)
            k = i * prm%NumCst(2)
            prm%rnode(i) = k
        end do
        ! ========================== right ========================== !
        if (prm%bcright .eq. 'd') then
        if (prm%bctop .eq. 'd') then
            if (prm%bcbottom .eq. 'd') then
            do i = 2,prm%NumCst(3)-1
                prm%qbc(j+(i-1)) = prm%rnode(i)
                prm%qbcval(j+(i-1)) = prm%PhysCst(4)
                !prm%qbcval(j+(i-1)) = 0D0                            ! QNODE 1
                !prm%qbcval(j+(i-1)) = (1D0-((prm%y(i)-prm%ymin)/(prm%ymax-prm%ymin))) ! QNODE
2

                !prm%qbcval(j+(i-1)) = ((prm%y(i)-prm%ymin)/(prm%ymax-prm%ymin))         !
QNODE 3
                !prm%qbcval(j+(i-1)) = 0D0                            ! QNODE 4
            end do
            j = j + prm%NumCst(3)-2
            else
            do i = 1,prm%NumCst(3)-1
                prm%qbc(j+i) = prm%rnode(i)
                prm%qbcval(j+i) = prm%PhysCst(4)
            end do
            j = j + prm%NumCst(3) - 1
            end if
        else if (prm%bcbottom .eq. 'd') then
            do i = 2,prm%NumCst(3)
                prm%qbc(j+(i-1)) = prm%rnode(i)
                prm%qbcval(j+(i-1)) = prm%PhysCst(4)
            end do
        j = j + prm%NumCst(3) - 1
        end if
        end if
        ! ========================== left ========================== !
        do i = 1,prm%NumCst(3)
            prm%lnode(i) = prm%rnode(i) - prm%neX
        end do
        if (prm%bcleft .eq. 'd') then
        if (prm%bctop .eq. 'd') then
            if (prm%bcbottom .eq. 'd') then
            ! All dirichlet case !
            do i = 2,prm%NumCst(3)-1
                prm%qbc(j+(i-1)) = prm%lnode(i)
                !prm%qbcval(j+(i-1))=(1D0-((prm%y(i)-prm%ymin)/(prm%ymax-prm%ymin)))!QNODE1
                prm%qbcval(j+(i-1))=prm%PhysCst(3)
                !prm%qbcval(j+(i-1))=0D0                                  !QNODE 2
                !prm%qbcval(j+(i-1))=0D0                                  !QNODE 3
                !prm%qbcval(j+(i-1))=((prm%y(i)-prm%ymin)/(prm%ymax-prm%ymin))    !QNODE 4
            end do
            j = j + prm%NumCst(3)-2
            else
            do i = 1,prm%NumCst(3)-1
                prm%qbc(j+i) = prm%lnode(i)
                prm%qbcval(j+i) = prm%PhysCst(3)
            end do
            j = j + prm%NumCst(3) - 1
            end if
        else if (prm%bcbottom .eq. 'd') then
            do i = 2,prm%NumCst(3)
                prm%qbc(j+(i-1)) = prm%lnode(i)
                prm%qbcval(j+(i-1)) = prm%PhysCst(3)
            end do
        j = j + prm%NumCst(3) - 1
        end if
        end if
        !
        prm%Nbc = j                ! Number of boundary points with Dirichlet type B.C.
        !
        end subroutine bcond
      allocate(sparse%A  (sparse%nonzero + 2*TheParam%Nbc)) !Allocating sparse matrix A -
      allocate(sparse%IRN(sparse%nonzero + 2*TheParam%Nbc)) !  taking into account the -
      allocate(sparse%JCN(sparse%nonzero + 2*Theparam%Nbc)) !  additional B.C. entries.
      call assembly (TheParam,sparse,bf)  ! Assembling sparse matrix
```

```
        subroutine assembly (par,sparse,bf)
            use ParamModule
            implicit none
            integer :: k,l,m,n
            Type (AIJ),target:: sparse
            Type (param), target:: par
            Type (BasFunc), target :: bf

            do k = 1, par%Tne  for all elements
                !
                do n = 1, par%NumCst(1)   ! for all node pairs
                    do m = 1, par%NumCst(1)
                    l = sparse%GML(k,n,m) ! the Global node identifier thing for supports, functions as an array locator
                    sparse%A(l)    = sparse%A(l) + bf%Aloc(k,n,m)
                    sparse%IRN(l)  = par%Lgm(k,n)
                    sparse%JCN(l)  = par%Lgm(k,m)
                    end do
                end do
            end do
            end subroutine assembly
        call lagmul (TheParam,sparse)       ! Extending sparse matrix with multipliers
        subroutine lagmul (prm,sp)
            ! Extend the global matrix, u_matrix and RHS_matrix with multipliers                !
            ! ================================================================================ !
            use ParamModule
            implicit none
            type (Param)   :: prm
            type (AIJ)     :: sp
            type (BasFunc) :: bf
            integer :: i
            sp%nbdof   = prm%Tnp
            sp%nonzero = count (sp%A /= 0D0)
            do i = 1, prm%Nbc
            sp%A(sp%nonzero + i) = 1D0
            sp%IRN(sp%nonzero + i) = prm%qbc(i)
            sp%JCN(sp%nonzero + i) = i + sp%nbdof
            end do
            sp%nonzero = count (sp%A /= 0D0)
            do i = 1, prm%Nbc
            sp%A(sp%nonzero + i) = 1D0
            sp%IRN(sp%nonzero + i) = i + sp%nbdof
            sp%JCN(sp%nonzero + i) = prm%qbc(i)
            end do
            sp%nonzero = count (sp%A /= 0D0)
            !
            end subroutine lagmul
        end subroutine Matrix_A
    call RHS (theParam,sparse,qd,bf)
    subroutine RHS (theParam,sparse,qd,bf)
        !-------------------------------------------------------------------------------!
        use ParamModule
        use quadrature
        implicit none
        Type (Param),target::theParam
        Type (AIJ),target:: sparse
        Type (quad),target::qd
        Type (BasFunc),target :: bf
        call calRHSloc (TheParam,qd,bf)  ! Local RHS entries for every element
            subroutine calRHSloc (par,qd,bf)
                !
                use ParamModule
                use quadrature
                implicit none
                type (Param)    :: par
                type (quad)     :: qd
                type (BasFunc)  :: bf
                integer          :: k,m,i,j
                double precision :: sr,dx,dy,xr,yr,a,b,c,d
                double precision :: xmin,xmax,ymin,ymax,xe,ye,sigma,pi
                pi = 3.14159265359
                allocate (bf%rhsLoc(par%Tne,par%NumCst(1)))
                a = ((par%xmax - par%xmin ) / 3D0) + par%xmin
                b = a + ((par%xmax - par%xmin ) / 3D0)
                c = ((par%ymax - par%ymin ) / 3D0) + par%ymin
                d = c + ((par%ymax - par%ymin ) / 3D0)
                do k = 1,par%Tne

                xmin = par%lex(k,1)
                xmax = par%lex(k,2)
```

```
            ymin = par%ley(k,1)
            ymax = par%ley(k,4)
            xe = (xmin + xmax) / 2D0
            ye = (ymin + ymax) / 2D0
            dx = xmax - xmin
            dy = ymax - ymin

            do m = 1,par%NumCst(1)
                do i = 1,par%NumCst(4)
                do j = 1,par%NumCst(4)
            !
                        xr = (dx/2.)*qd%quad_x0(i) + (dx/2.)
                        yr = (dy/2.)*qd%quad_x0(j) + (dy/2.)
            !           basis functions
                        bf%f(1) = (1. - xr/dx) * (1. - yr/dy)
                        bf%f(2) = (xr/dx) * (1. - yr/dy)
                        bf%f(3) = (xr/dx) * (yr/dy)
                        bf%f(4) = (1. - xr/dx) * (yr/dy)
                    !sr =  sin(0.5*pi*xe)*sin(0.5*pi*ye)      !par%PhysCst(5)
            sr = 0
            if(xe.ge.-1.AND.xe.le.1.AND.ye.ge.0.7.AND.ye.le.1) then
            sr  = 1D0
            end if
            if(xe.ge.-1.AND.xe.le.1.AND.ye.ge.-1.AND.ye.le.-0.7) then
            sr  = 1D0
            end if
                    bf%rhsLoc(k,m)=bf%rhsLoc(k,m)+((dx*dy)/4.)*sr*qd%quad_w(i)*qd%quad_w(j)*bf%f(m)
                end do
                end do
            !end if
            end do
            end do
            !
            end subroutine calRHSloc
        call GRHS (sparse,TheParam,bf)   ! Global RHS entries
            subroutine GRHS (sparse,par,bf)
                use ParamModule
                implicit none
                Type (AIJ),target      :: sparse
                Type (param), target   :: par
                Type (BasFunc), target :: bf
                integer                :: k,m,n,j,i,j1,j2
                !
                allocate(sparse%RHS(par%Tnp + par%Nbc))
                !
                do k = 1, par%Tne                          ! Storing RHS matrix
                do n = 1, par%NumCst(1)
                    m = par%Lgm(k,n)
                        sparse%RHS(m) = sparse%RHS(m)  +  bf%rhsLoc(k,n)
                end do
                end do
                j = par%Tnp                                ! Storing B.C. (Dirichlet) values
                j1 = par%Nbc                               ! on RHS matrix.
                do i = j+1,j+j1
                j2 = i - j
                sparse%RHS(i) = par%qbcval(j2)
                end do
                end subroutine GRHS
            end subroutine RHS
        call solve(sparse,TheParam)
            subroutine solve (sparse,par)
                use ParamModule
                TYPE(DMUMPS_STRUC) :: id
                TYPE(AIJ),INTENT(INOUT):: sparse
                TYPE(Param),INTENT(INOUT):: par
                integer :: ierr, i
                integer :: m,n
                integer :: j,k,l
                print *, ' calling external MUMPS Solver...'
            !   id%ICNTL(1) = 0
            !   id%ICNTL(2) = 0
            !   id%ICNTL(3) = 0
            !   id%ICNTL(4) = 0
                print *, 'Initializing ...'
                ! ------------------------------------------- !
                ! initialize mumps                            !
                id%SYM = 0                                    !
                ! Host working                                !
                id%PAR = 1                                    !
```

Basis functions as seen before

Sources

$-1 < x < 1$ and $0.7 < y < 1$ and $-1 < x < -0.7$

Source terms

Gaussian weights

Generates the local source terms

Global RHS entries

Storing RHS matrix

Storing B.C. (Dirichlet) values on RHS matrix.

Sparse RHS = ({x...x} , {y...y})

x = source term array numbers true global node number

Summary so far

we have Sparse %A = { .... } numerical value

(1) k n m

true 1 array index

16 bit number representing the node pair

eg index 6 means (1,2,2), which is global node pairs 2 and 2

But how do we find these pairs? well that's what IRN and JCN are for!

So under 6 of IRN and JCN is 2, so yes!

IRN returns the n node, JCN returns the M node.

I Row nodes

J Column nodes

indices

This is an example of a sparse matrix

```
call solve(sparse,TheParam)
    subroutine solve (sparse,par)
        use ParamModule
        TYPE(DMUMPS_STRUC) :: id
        TYPE(AIJ),INTENT(INOUT):: sparse
        TYPE(Param),INTENT(INOUT):: par
        integer :: ierr, i
        integer :: m,n
        integer :: j,k,l
        print *, 'calling external MUMPS Solver...'
        !   id%ICNTL(1) = 0
        !   id%ICNTL(2) = 0
        !   id%ICNTL(3) = 0
        !   id%ICNTL(4) = 0
        print *, 'Initializing ...'
        ! ------------------------------------------------ !
        ! initialize mumps                                 !
        id%SYM = 0                                         !
        ! Host working                                     !
        id%PAR = 1                                         !
        ! Initialize an instance of the package            !
        id%JOB = -1                                        !
        CALL DMUMPS(id)                                    !
        ! ------------------------------------------------ !
        id%N    = sparse%nbdof + par%Nbc
        id%NZ   = sparse%nonzero
        Allocate (id%RHS (sparse%nbdof + par%Nbc))
        Allocate (id%IRN (sparse%nonzero))
        Allocate (id%JCN (sparse%nonzero))
        Allocate (id%A   (sparse%nonzero))
        print *, 'Reading the matrix and the RHS...'
        ! mounting sparse matrix values
        do i = 1,sparse%nonzero
            id%A(i)   = sparse%A(i)
            id%IRN(i) = sparse%IRN(i)
            id%JCN(i) = sparse%JCN(i)
        end do
        ! mounting rhs matrix values
        do i = 1,sparse%nbdof + par%Nbc
            id%RHS(i) = sparse%RHS(i)
        end do
        print *, 'Solving...'
        ! ------------------------------------------------ !
        id%JOB = 6                                         !
        ! upper memory bound for MUMPS                     !
        id%icntl (23) = par%NumCst(5)                      !
        ! id%icntl (6) = 0 ! no permutation                !
        ! id%icntl (8) = 8 ! no scaling                    !
        ! error analysis                                   !
        id%ICNTL(11) = 1                                   !
        !id%ICNTL(11) = 0                                  !
        ! scaling provided by user                         !
        ! id%ICNTL(8) = -1                                 !
        CALL DMUMPS(id)                                    !
        ! ------------------------------------------------ !
        ! write solution
        do i = 1,sparse%nbdof + par%Nbc
            sparse%RHS(i) = id%RHS(i)
        end do
        ! cleanup
        deallocate (id%IRN)
        deallocate (id%JCN)
        deallocate (id%A  )
        deallocate (id%RHS)
        print *, 'dumping memory...'
        ! ------------------------------------------------ !
```

Matlab study
mare Program

Type (Parameters)
x: int
y: int.

Subroutine add (Parameters)
x+y

Call add (Param)

```
        id%JOB = -2                                             !
        CALL DMUMPS(id)                                         !
        ! ---------------------------------------------------- !
        end subroutine solve
call reference (theParam,sparse)
    subroutine reference (par,spar)
        use ParamModule
        implicit none
        double precision :: p,pi,a,b
        Type (Param), target :: par
        Type (AIJ), target :: spar
        integer :: i,j
        !
        a = par%xmax
        b = par%ymax
        !
        p = 5
        pi = 3.14159265359
        allocate (par%uex(spar%nbdof))
        !
        do i = 1, spar%nbdof
        par%uex(i) = sin(p*pi*par%xg(i)/a)*sinh(p*pi*par%yg(i)/a)/sinh(p*pi*b/a)
        end do
        !
        call ErrorEst (par,spar)
            subroutine ErrorEst (par,spar)
                use ParamModule
                implicit none
                type (Param), target :: par
                type (AIJ), target :: spar
                integer :: i,j,k
                double precision :: uh,ue,x,uer
                ! L_inf Error estimate
                par%uerr = 0D0
                uer = 0D0
                do i = 1, spar%nbdof
                uh = spar%RHS(i)
                ue = par%uex(i)
                x = (abs(uh - ue))**2
                uer = uer + x
                end do
                par%uerr = sqrt(uer)
                end subroutine ErrorEst
        !
        end subroutine reference
call output (TheParam,sparse,bf)
    subroutine output (TheParam,sparse,bf)
        use ParamModule
        implicit none
        integer ::  i,k,l,m
        type (param) :: TheParam
        type (AIJ) :: sparse
        type (BasFunc) :: bf
        !
        open(unit = 40, file = 'local_matrix.txt',status = 'unknown')
        open(unit = 60, file = 'ref',form = 'unformatted')
        open(unit = 50, file = 'SOL.tec')
        open(unit = 80, file = 'sparse_mat.txt',status = 'unknown')
        open(unit = 90, file = 'EXACT.tec',status = 'unknown')
        ! ============== !
        ! writing block !
        ! ============== !
        print *, 'writing results...'
        !write(80,*) '============================================='
        !write(80,*) '               sparse (AIJ) matrix           '
        !write(80,*) '============================================='
        !write (80,*) 'i : A(i) : I(i) : J(i)'
        !write(80,*) '---------------------------------------------'
        !do l = 1,sparse%nonzero
        !   write (80,'(I10,A,F10.4,I10,I10)') l,' ',sparse%A(l),sparse%IRN(l),sparse%JCN(l)
        !end do
        ! solution
        write(50,*) 'variables = "x","y","u"'
```

```
        write(50,*) 'zone t ="solution",I=',TheParam%NumCst(2),',J=',TheParam%NumCst(3)
        do i = 1,sparse%nbdof
            write(50,*) TheParam%xg(i),TheParam%yg(i),sparse%RHS(i)
        end do
        close (50)

        write (60) sparse%RHS
        close (60)
        ! exact reference
        !write(90,*) 'variables = "x","y","u"'
        !write(90,*) 'zone t ="solution",I=',TheParam%NumCst(2),',J=',TheParam%NumCst(3)
        !do i = 1,sparse%nbdof
        !   write(90,*) TheParam%xg(i),TheParam%yg(i),sparse%rhs(i)!TheParam%uex(i)
        !end do
        !
        !write(60,*) '==================='
        !write(60,*) '  Map_loc  '
        !write(60,*) '==================='
        !do k = 1,TheParam%Tne
        !   write(60,*)  k,(TheParam%Lgm(k,l),l = 1,TheParam%NumCst(1))
        !end do
        !
        write(40,*) '===================='
        write(40,*) 'Local matrix'
        write(40,*) '===================='
        do k = 1, TheParam%NumCst(1)
            write(40,'(10000000000000(F20.5))') (bf%Aloc(1,k,l), l = 1,TheParam%NumCst(1))
        end do
        write(40,*) '===================='
        write(40,*) 'Local RHS           '
        write(40,*) '===================='
        do k = 1, TheParam%NumCst(1)
            write(40,*) bf%rhsLoc(1,k)
        end do
        !write(40,*) '===================='
        !write(40,*) '      GML           '
        !write(40,*) '===================='
        !do k = 1,TheParam%Tne
        !   do l = 1,TheParam%NumCst(1)
        !      do m = 1,TheParam%NumCst(1)
        !         write(40,*) k,l,m,sparse%GML(k,l,m)
        !      end do
        !   end do
        !end do
        end subroutine output


write(*,*) '==================== the L2 error estimation ========================='
write(*,'(A,I10,A,F8.4,A)')'for N=',theParam%Tnp,' and h=',theParam%h(1),''
write(*,'(A,F10.5)')'L2_error=',theParam%uerr
write(*,*) '======================================================================='
end program
```