周吕文 201128000718065
物理学院 20110308 班
计算报告
4/15/2012

不可压缩二维 N-S 方程的 SIMPLE 解法

1 引言

SIMPLE 算法全称是 "Semi-Implicit Method for Pressure-Linked Equations", 意思是求解压力耦合的质量/动量/能量传递方程的半隐式方法. SIMPLE 算法自 1972 年由 D.B.Spalding 和 S.V.Patankar 提出以来在世界各国计算流体力学及计算传热学界得到了广泛的应用, 它实际上已经成为许多工程流动, 传热以及反应体系的数值模拟的最重要的方法. 许多商业 CFD 软件, 如 cfx 与 fluent, 其核心也都基于 SIMPLE 算法.

SIMPLE 算法适合求解不可压流场的数值问题, 也可用于求解可压流动. 它的核心是采用 "猜测 -修正" 的过程. 基本思想: 对于给定的压力场 (它可以是假定值或是上一次迭代计算所得到的结果), 求解离散形式的动量方程, 得出速度场. 因为压力场是假定的或不精确的, 这样得到的速度场一般不满足连续方程, 因此, 必须对给定的压力场加以修正. 修正的原则是: 与修正后的压力场相对应的速度场能满足这一迭代层次上的连续方程离散形式. 据此原则, 我们把由动量方程的离散形式所规定的压力与速度的关系代入连续方程的离散形式, 从而得到压力修正方程, 由压力修正方程得出压力修正值. 接着, 根据修正后的压力场, 求得新的速度场. 然后检查速度场是否收敛. 若不收敛, 用修正后的压力值作为给定的压力场, 开始下一层次的计算, 直至收敛为止.

2 不可压缩二维 N-S 方程

对于二维的不可压缩问题, 其 N-S 方程描述为

$$u_t + p_x = -\left(uu\right)_x - \left(uv\right)_y + \frac{1}{Re}\left(u_{xx} + u_{yy}\right) \tag{1}$$

$$v_t + p_y = -\left(uv\right)_x - \left(vv\right)_y + \frac{1}{Re}\left(v_{xx} + v_{yy}\right) \tag{2}$$

$$u_x + v_y = 0 \tag{3}$$

其中 u, v 分别是 x 和 y 方向的速度, p 为压强, Re 是雷诺数.

3 SIMPLE 算法

根据 SIMPLE 算法, 首先本文需要设定一个初始压力场 $(P^*)^n$, 以及初始速度场 $(U^*)^n$ 和 $(U^*)^n$, 然后根据动量方程求出 $(U^*)^{n+1}$ 和 $(U^*)^{n+1}$, 再通过解压力校正方程

$$\nabla^2 p' = \frac{1}{\Delta t}(\nabla \cdot V)$$

来求得压力校正值 p', 将猜测值 p^* 与校正值 p' 合成. 然后得到修正后的压力场和速度场. SIMPLE 算法的步骤如图1所示.

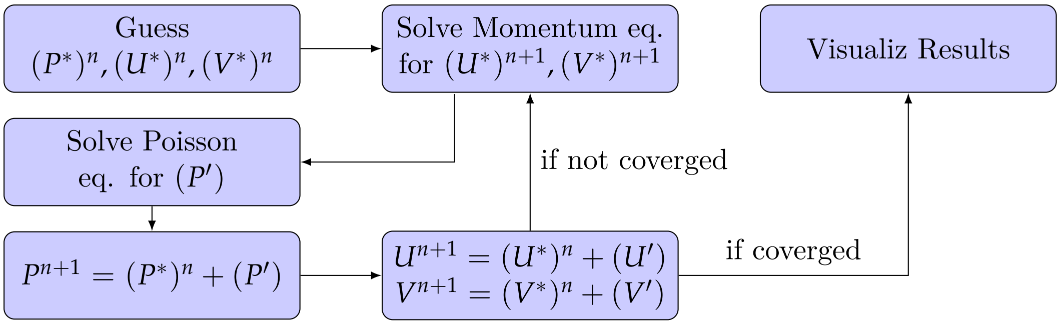

图 1: SIMPLE 算法的步骤

3.1 交错网格

为了对不可压缩二维 N-S 方程离散差分, 本文选则如图3所示的矩形交错网格. 把速度分量和压力分量分别设置在三套不同的网格单元上. 其中压力设置在主控制单元的中心. 相应的水平方向速度 u 设置在主控制单元的左右边界上, 相应的竖直方向速度 v 设置在主控制单元的上下边界上.

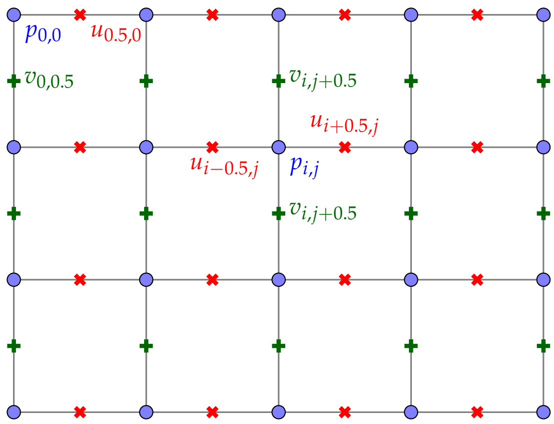

图 2: 交错网格

对于本文后面将要计算的一些算例, 由于固壁上的速度都为零, 所以本文将速度设置在边界上, 即水平速度设置在竖直边界上, 竖直速度设置在水平边界上. 考虑边界, 在物理边界外层设计半层虚拟网格. 比如在交错凹管道中的流动问题中, 其网格设置的示意图见图3 从图3中可见,

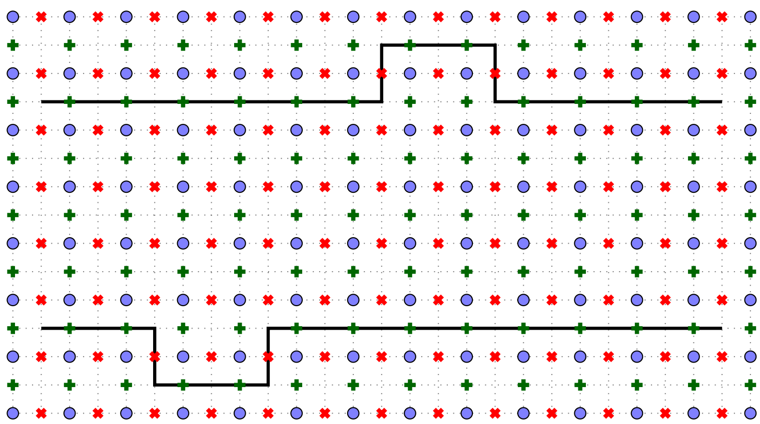

图 3: 交错凹管道中的流动问题的网格设置:‘·’ 表示压力 p,‘x’ 表示速度 u,‘+’ 表示速度 v

在水平方向上速度 v 和压强 p 的节点数相等 (本文设为 n_x+1), 并且都比 u 多一个节点; 在竖直方向上, 速度 u 和压强 p 的节点数相等 (本文设为 n_y+1), 并且都比 v 多一个节点. 因此速度 u, v 及压强的节点数分别为

$$n_x \times (n_y+1),\ \ (n_x+1)\times n_y,\ \ (n_x+1)\times(n_y+1)$$

基于这种简单的交差网格的设置, 我们可以得到二阶精度的差分. 下面来讨论本文的差分格式.

3.2 差分格式

对于图1中的 SIMPLE 算法的步骤, 本文只需要对三个方程进行离散, 首先本文对动量方程式 (1) 和式 (2) 进行离散. 在动量方程的离散中, 各项的离散形式和精度见表1. 动量方程式 (1)

表 1: 差分形式

导数项	差分形式	导数项	差分形式	差分类型
u_t	$\frac{u^{n+1}-u^n}{\Delta t}$	v_t	$\frac{v^{n+1}-v^n}{\Delta t}$	forward, $O(h)$
u_x	$\frac{u_{i+i,j}-u_{i-1,j}}{\Delta x}$	v_y	$\frac{v_{i+i,j}-v_{i-1,j}}{\Delta y}$	central, $O(h^2)$
u_{xx}	$\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\Delta x)^2}$	v_{xx}	$\frac{v_{i+1,j}-2v_{i,j}+v_{i-1,j}}{(\Delta x)^2}$	central, $O(h^2)$
u_{yy}	$\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\Delta y)^2}$	v_{yy}	$\frac{v_{i,j+1}-2v_{i,j}+v_{i,j-1}}{(\Delta y)^2}$	central, $O(h^2)$
$(uu)_x$	$u_{i,j}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}$	$(vv)_y$	$v_{i,j}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta y}$	central, $O(h^2)$
$(uv)_y$	$u_{i,j}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta y}+v_{ij}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta y}$	$(uv)_x$	$u_{i,j}\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta x}+v_{ij}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}$	central, $O(h^2)$
p_x	$\frac{p_{i+1,j}-p_{i,j}}{\Delta x}$	p_y	$\frac{p_{i,j+1}-p_{i,j}}{\Delta y}$	forward, $O(h)$

和式 (2) 在图3所示的网格上的离散形式如下

$$u_{i,j}^{n+1} = u_{i,j}^{n} + dt\Big[-A-B+\frac{1}{\mathrm{Re}}(C+D)-\frac{p_{i+1,j}-p_{i,j}}{\Delta x}\Big]$$

$$v_{i,j}^{n+1} = v_{i,j}^{n} + dt\Big[-E-F+\frac{1}{\mathrm{Re}}(G+H)-\frac{p_{i,j+1}-p_{i,j}}{\Delta y}\Big]$$

其中

$$A = u_{i,j}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}$$

$$B = \frac{1}{4}\big(v_{i,j}+v_{i+1,j}+v_{i,j-1}+v_{i+1,j-1}\big)\frac{u_{i,j+1}-u_{i,j-1}}{2\Delta y}$$

$$C = \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\Delta x)^2}$$

$$D = \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\Delta y)^2}$$

$$E = v_{i,j}\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}$$

$$F = \frac{1}{4}\big(u_{i-1,j}+u_{i-1,j+1}+u_{i,j}+u_{i,j+1}\big)\frac{v_{i+1,j}-v_{i-1,j}}{2\Delta x}$$

$$G = \frac{v_{i+1,j}-2v_{i,j}+v_{i-1,j}}{(\Delta x)^2}$$

$$H = \frac{v_{i,j+1}-2v_{i,j}+v_{i,j-1}}{(\Delta y)^2}$$

对于 Poisson 方程的求解, 本文利用四点迭代法求压力. 对于每一步 Poisson 方程的迭代, 其具体形式如下

$$p'_{i,j} = \frac{1}{a}\left[b\left(p'_{i+1,j} + p'_{i-1,j}\right) + c\left(p'_{i,j+1} + p'_{i,j-1}\right) + d\right] \tag{4}$$

其中各项分别为下

$$b = -\frac{\Delta t}{\Delta x^2},\ c = -\frac{\Delta t}{\Delta y^2},\ a = -2(a+b)$$

对于式样 (4) 的迭代方式非常简单. 每一步迭代, 本文都对每一个压强点用上式计算迭代后的值, 一步迭代完成后, 检查一下同一压强点上的迭代前后值相差的最大值是否为 $\varepsilon = 0$(实际计算过程中,ε 取一个非常小的值, 当迭代前后的 p' 相差的最大值小于 ε 即认为迭代完成)

3.3 边界条件

在交错网格中, 边界处理相对于非交错网格较为复杂, 一些量的节点恰好在边界上, 而另一些节点则分布在边界两侧, 因此在施加边界条件时要非常小心, 本文对速度应用狄利克雷边界条件 (Dirichlet boundary condition), 压强施加诺伊曼边界条件 (Neumann boundary condition). 对于速度恰好在边界上的情况, 应用边界条件时真接将这些速度设为边界上的速度 U_b(对于固壁边界条件, $U_b = 0$), 比如在左右边界上的 u 及上下边界上的 v. 而对于速度点分布在边界两边的情况, 比如在上下边界的 v 和左右边界的 v, 其边界条件为

$$\frac{u_{i,j} + u_{i,j+1}}{2} = U_b \impliedby u_{i,j} = 2U_b - u_{i,j+1}\ \text{或}\ u_{i,j+1} = 2U_b - u_{i,j}$$

对于边界上的压力的边界条件, 压力在边界法向上的导数 $\partial p/\partial \mathrm{n}$ 由边界两边对应的两个压强点定义. 比如对于左边界, 诺伊曼边界条件为

$$\frac{p_{i,j+1} - p_{i,j}}{dy} = 0 \implies p_{i,j+1} = p_{i,j}$$

实际上, 在本文的交错网格中, 固壁边界外的压力点并不参与到边界内速度的计算中, 因此在固壁上并不需要对压强施加边界条件. 只在最后结果的后处理中将压强平均到网格节点上时需要.

4 算例

本文通过 Fortran95 编写了针对一般情况下不可压缩二维 N-S 方程的 SIMPLE 解法, 程序见附录. 只需在程序中给出相应的边界及边界条件 (仅限于水平, 竖直, 及 45 度边界), 即可计算出相应的稳态结果. 用该程序很容易测试不同的情况. 应用该程序, 本文不仅测试了本次作业指定的交错凹管道中的流动问题, 还测试了不同是形状的通道, 以及通道中有不同形状和数量的障碍物的情况, 下面列举出部分本文测试的算例.

4.1 交错凹管道中的流动问题

交错凹管道中的流动问题是本次作业指定需要完成的题, 该问题初始设置如图4所示, 在一条高为 2, 长为 6 的的通道中有两个凹槽. 管道的左端为初始入口, 入口处的水平速度 u 如图4中标注, $v = 0$, $p = 1$. 右端开口为出口边界条件.

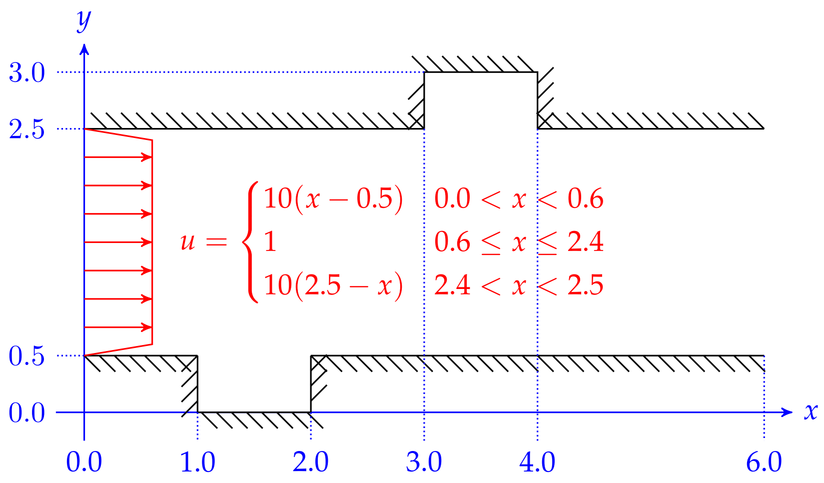

图 4: 交错凹管道中的流动问题

通过对图4所示的交错凹管道中的流动问题的 SIMPLE 计算, 本文得到了该问题的数值解, 即稳态的速度场和压力场. 图5和图6是计算得到的速度场. 从速度场可以看出, 在通道内近边界处的水平速度非常用小, 中间较大, 这说与无滑移边界条件完全符合. 另外在凹槽内, 水平速度非常用小. 在角点处, 速度 v 较大, 其它区域, 速度 v 几乎为零. 图7交错凹管道中的流动问题的

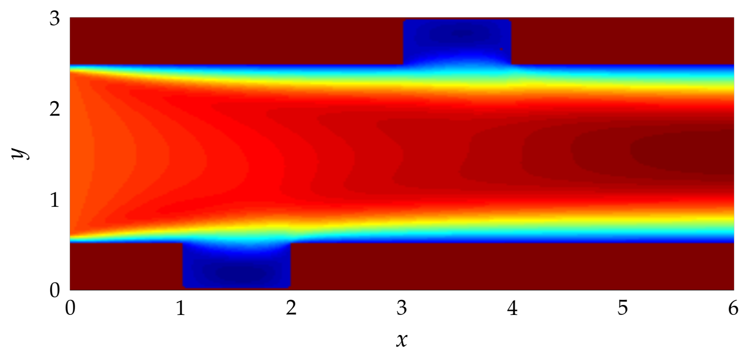

图 5: 交错凹管道中的流动问题的 u 速度场

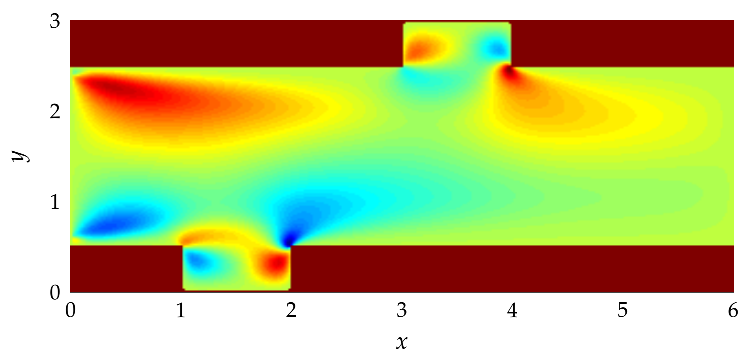

图 6: 交错凹管道中的流动问题的 v 速度场

压力场及流线图, 从该图中可以看出, 在两个凹槽内形成了两个涡, 中间流线基本场均水平, 在出口处, 流线完全水平, 压力均匀.

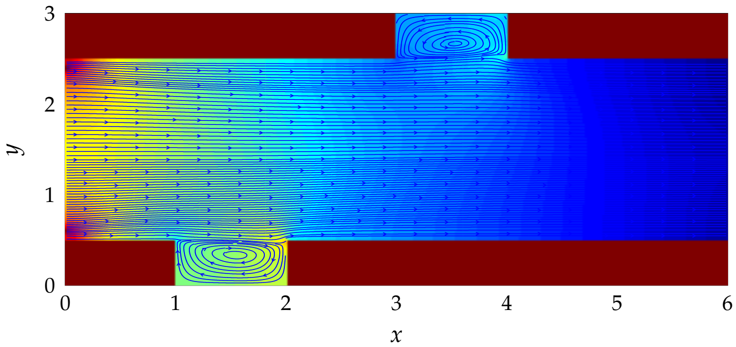

图 7: 交错凹管道中的流动问题的压力场及流线图

4.2 其它算例

交错凹管道中的流动问题外, 本文还对奇性方腔流, 弯折管道, 对称缩腰管道中的流动以及流场中有方块, 三角等一个或多个障碍物的情况进行了测式. 下面给出了部分算例的测试结果.

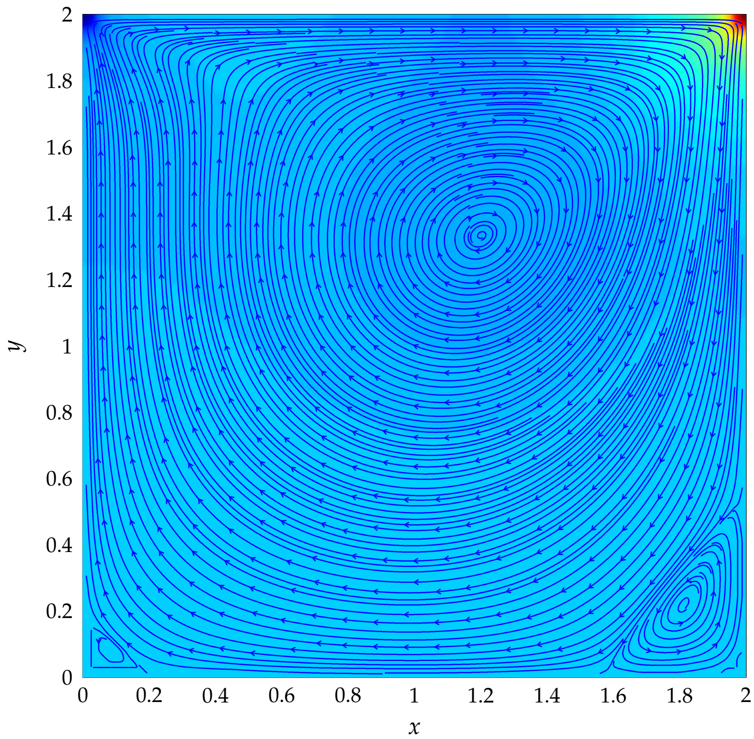

图 8: 奇性方腔流

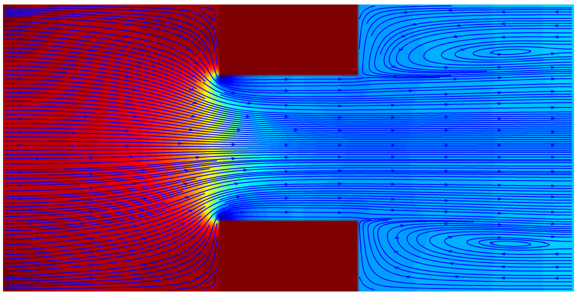

图 9: 对称缩腰管道

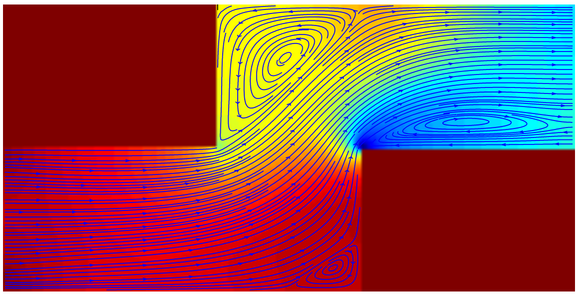

图 10: 弯折管道

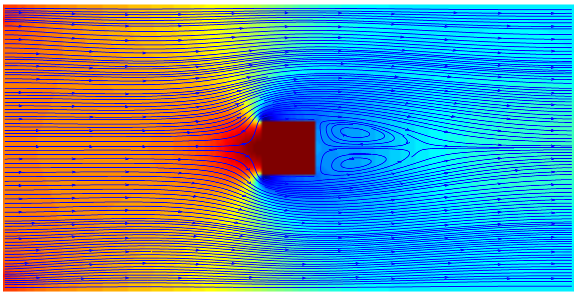

图 11: 流场中一个方块情况

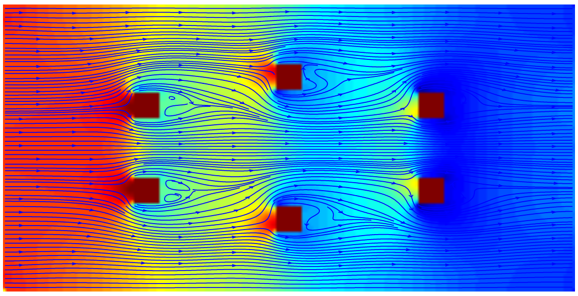

图 12: 流场中多个方块情况

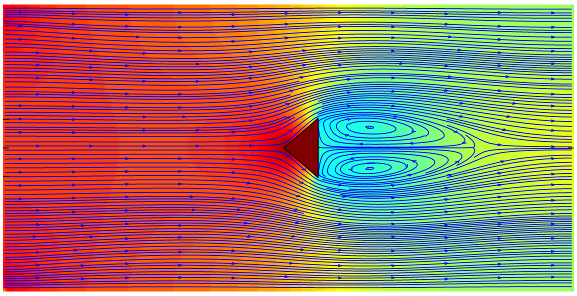

图 13: 流场中三角形情况

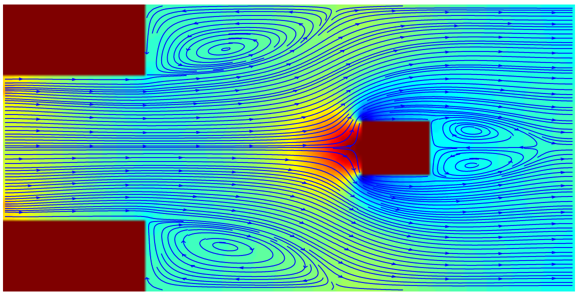

图 14: 扩张流中方块情况

参考文献

[1] 张德良, 计算流体力学教程. 高等教育出版社. 北京, 2011.

[2] Maciej Matyka. Solution to two-dimensional Incompressible Navier-Stokes Equations with SIMPLE, SIMPLER and Vorticity-Stream Function Approaches. Driven-Lid Cavity Problem: Solution and Visualization.

[3] Benjamin Seibold. A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains

[4] SIMPLE algorithm. http://en.wikipedia.org/wiki/SIMPLE_algorithm

附录

A CFD2dSimple.f90

```fortran
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module parameters
 implicit none
 integer, parameter      :: o      = 8       ! Double Precision
 real(o), parameter      :: dt     = 0.005   ! time step
 real(o), parameter      :: Re     = 100     ! Reynolds number
 real(o), parameter      :: dx     = 0.02    ! x - spatial step size
 real(o), parameter      :: dy     = 0.02    ! y - spatial step size
 real(o), parameter      :: toleror= 1e-4    ! tolerance of error
 real(o), parameter      :: prec   = 1e-3    ! precision of poisson eq.
end module parameters
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module variables
 use parameters; implicit none
 integer                 :: i, j, k          ! loop index
 real(o)                 :: resmax           !
 real(o), allocatable    :: u(:,:)           ! Horizontal velocity
 real(o), allocatable    :: v(:,:)           ! vertical velocity
 real(o), allocatable    :: p(:,:)           ! pressure
 real(o), allocatable    :: uStar(:,:)       ! u*
 real(o), allocatable    :: vStar(:,:)       ! v*
 real(o), allocatable    :: pCorr(:,:)       ! p'
 real(o)                 :: gx     = 0       ! External force
 real(o)                 :: gy     = 0       ! External force
 integer                 :: caseIndex        ! computational case
 character               :: caseName*20      ! Difference scheme
 real(o)                 :: t = 0            ! time
end module variables
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module boundaryclass
! Defined boundary class
 use parameters; implicit none
 type bound
    real                :: x(2) , y(2)       ! The endpoints of the bound
    integer             :: i(2) , j(2)       ! i = x/dx + 1; j = y/dy + 1
    integer             :: side              ! side: 1 right ;  2 left
                                             !       3 bottom;  4 top
    integer             :: typeU, typeV      ! type: 1 constant;
                                             !       2 constant derivative
    real                :: Uval , Vval       ! the velocity value of bound
 end type bound

 contains
 subroutine NewBound(this, x, y, side, types, values)
  type(bound)           :: this              !
  real    , dimension(2) :: x,y              ! boundary edge:
                                             !          (x1,y1), (x2,y2)
  integer               :: side              ! side: left/top/...
  integer, dimension(2) :: types             ! types of boundary condition
  real    , dimension(2) :: values           ! boundary parameters of u&v
```

```fortran
  this%side  = side
  this%x     = x;                this%y    = y
  this%i     = aint(x/dx)+1;     this%j    = aint(y/dy)+1
  this%typeU = types(1);         this%Uval = values(1)
  this%typeV = types(2);         this%Vval = values(2)
 end subroutine NewBound
end module boundaryclass
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module boundary
 use  boundaryclass; implicit none
 integer                  :: Nbound           ! number of boundarys
 type(bound),allocatable:: bounds(:)          ! boundarys
 logical, allocatable    :: uin(:,:)          ! if u(i,j) in the boundary
 logical, allocatable    :: uon(:,:)          ! if u(i,j) in the boundary
 logical, allocatable    :: vin(:,:)          ! if v(i,j) in the boundary
 logical, allocatable    :: von(:,:)          ! if v(i,j) in the boundary
 logical, allocatable    :: pin(:,:)          ! if p(i,j) in the boundary
 logical, allocatable    :: pon(:,:)          ! if v(i,j) in the boundary

 integer                 :: imin,  imax       ! imin = min(i1,i2)
 integer                 :: jmin,  jmax       ! jmax = max(j1,j2)
 integer                 :: side              ! side: left/top/...
 integer                 :: typeU, typeV      ! types of boundary
 real(o)                 :: Uval,  Vval       ! boundary parameters
end module boundary
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module grid
 use parameters
 implicit none
 integer                  :: nx, ny           ! Number of grid points
 real(o)                  :: Lx, Ly           ! Length of domain
 real(o), allocatable     :: xp(:,:), yp(:,:)! pressure points
 real(o), allocatable     :: xu(:,:), yu(:,:)! Horizontal velocity points
 real(o), allocatable     :: xv(:,:), yv(:,:)! vertical velocity points
end module grid
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
module process
 use parameters; implicit none
 real(o)                  :: resmax0 = 0      ! The initial resmax
 real(o)                  :: redfac           ! Reduction factor
 integer                  :: nbarmax = 50     ! the length of the bar
 integer                  :: nbar = 0         ! Initial length
end module process

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program CFD2Dsimple
 use parameters; use grid; use boundaryclass; use boundary; use variables
 implicit none

 call Initialization()
 call geometry()
 call InitGrid()
 call boundaryConditions()

 ! Check if solution coverges: If maximum change of pressure on a grid is
 !                    bigger than error,  we continue iterative process.
```

```fortran
 resmax = 2*toleror
 do while(resmax>toleror)

    call explicitEuler()
    call PresProj()
    call boundaryConditions()

    t = t + dt
        call progressBar()
 end do
 call output()


end program CFD2Dsimple
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine Initialization()
 use parameters; use variables
 implicit none

 write(*,*),"--Solution to 2D Incompressible NS Equations with SIMPLE--"
 write(*,*),"     Copyrhigt by Zhou Lvwen: zhou.lv.wen[at]gmail.com    "
 write(*,*)

 write(*,*),"---------------- computational parameters ----------------"
 write(*, '("  -Reynolds number   :", f7.2)') Re
 write(*, '("  -spatial step size :", f5.2)') dx
 write(*, '("  -time step         :", f7.4)') dt
 write(*,*)
 write(*,*),"------------------- computational case -------------------"
 write(*,*)," - case 0: Default case                          "
 write(*,*)," - case 1: Flow over square                      "
 write(*,*)," - case 2: Flow over triangle                    "
 write(*,*)," - case 3: Flow over squares                     "
 write(*,*)," - case 4: Driven Cavity                         "
 write(*,*)," - case 5: Flow in contraction channel           "
 write(*,*)," - case 6: Flow in orthogonal channel            "
 write(*,*)," - case 7: Suddenly Expanded & square            "
 write(*,'(A,$)'), ' Please select computational case(0-7): '
 read(*,*) caseIndex
end subroutine Initialization
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine geometry()
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Default case:
!   y
!  /|\
!   y4 ............................. ||ZZZZZZZZ||.....................
!   |                                ||        ||                    :
!   |                                ||        ||                    :
!   y3 .||ZZZZZZZZZZZZZZZZZZZZZZZZZZZ||        ||ZZZZZZZZZZZZZZZZZZZ||
!   |   ||\                           :         :                   ||
!   |   || \                          :         :                   ||
!   |   ||->\                         :         :                   ||
!   |   ||   |      / 10y      ,  if 0.0 <=y<= 1.0              du/dx = 0
!   |   ||-->|  u = | 1        ,  if 0.1 <=y<= 1.0                  ||
!   |   ||   |      \ 10(2.0-y),  if 1.9 <=y<= 2.0              dv/dx = 0
```

```
!   |   ||->/                         :         :                   ||
!   |   || /                          :         :                   ||
!   |   ||/                           :         :                   ||
!   y2 .||ZZZZZZZZZ||        ||ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ||
!   |   :          ||        ||       :         :                    :
!   |   :          ||        ||       :         :                    :
!   y1 .:......... ||ZZZZZZZZ||......::.........:....................:
!   |   :          :         :                                          x
!  -+---x1---------x2--------x3------x4--------x5-------------------x6-->
!
! X:  x1 = 0.0,  x2 = 1.0,  x3 = 2.0,  x4 = 3.0,  x5 = 4.0,  x6 = 6.0
! Y:  y1 = 0.0,  y2 = 0.5,  y3 = 2.5,  y4 = 2.5
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                  !
!  case 1: Flows over square       !    case 2: Flows over triangle
!                                  !
!  |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|  !   |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|
!  :.                           :  !   :.                           :
!  :..                          :  !   :..             /|           :
!  :...        |ZZZZZZ|         :  !   :...           /X|           :
!  :....       |ZZZZZZ|         :  !   :....         /XX|           :
!  :....       |ZZZZZZ|         :  !   :....         \XX|           :
!  :...        |ZZZZZZ|         :  !   :...           \X|           :
!  :..                          :  !   :..             \|           :
!  :.                           :  !   :.                           :
!  |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|  !   |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|
!                                  !
!                                  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                  !
!  case 3: Flows over squares      !   case 4: Driven Cavity
!                                  !
!  |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|  !   ||--->---->---->---->---->--||
!  :.                           :  !   ||           ____           ||
!  :..            |x|           :  !   ||         /      \         ||
!  :...  |X|              |x|   :  !   ||        /        \        ||
!  :....                        :  !   ||       (          )       ||
!  :...  |x|              |x|   :  !   ||                 /        ||
!  :..            |x|           :  !   ||           /____/         ||
!  :..                          :  !   ||           \              ||
!  :.                           :  !   ||                          ||
!  |ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ|  !   ||ZZZZZZZZZZZZZZZZZZZZZZZZZz||
!                                  !
!                                  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                  !
!  case 5: contraction channel     !    case 6: orthogonal channel
!                                  !
!  |ZZZZZZZZ|          |ZZZZZZZZ|  !             |ZZZZZZZZZZZZZZZZZZ|
!  :.     |Z|          |Z|      :  !             |Z|                :
!  :..    |ZZZZZZZZZZZZZZ|      :  !             |Z|   ___________\ :
!  :...                         :  !             |Z|   |          / :
!  :....                        :  !             |Z|   |            :
!  :....                        :  !   |ZZZZZZZZZZZ|   |   |ZZZZZZZZ|
!  :....                        :  !   :.              |   |Z|
!  :...                         :  !   :..     ________|   |Z|
```

```
!   :..     |ZZZZZZZZZZZZZZ|       :  !   :..                     |Z|
!   :.      |Z|            |Z|     :  !   :.                      |Z|
!   |ZZZZZZZZ|             |ZZZZZZZZ| !   |ZZZZZZZZZZZZZZZZZZZZZZZZZZ|
!                                     !
!                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 use grid; use boundary; use variables
 implicit none
 select case(caseIndex)
    case(1);          caseName = 'Flows over square'
       Lx = 4.0;      Ly = 2.0;     Nbound = 8
       allocate(bounds(Nbound))
       call NewBound(bounds(1 ), [0.0,0.0], [0.0,2.0], 2, [3,2], [0.,0.])
       call NewBound(bounds(2 ), [0.0,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
       call NewBound(bounds(3 ), [4.0,4.0], [2.0,0.0], 1, [2,2], [0.,0.])
       call NewBound(bounds(4 ), [4.0,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

       call NewBound(bounds(5 ), [1.8,1.8], [0.8,1.2], 1, [1,1], [0.,0.])
       call NewBound(bounds(6 ), [1.8,2.2], [1.2,1.2], 3, [1,1], [0.,0.])
       call NewBound(bounds(7 ), [2.2,2.2], [1.2,0.8], 2, [1,1], [0.,0.])
       call NewBound(bounds(8 ), [2.2,1.8], [0.8,0.8], 4, [1,1], [0.,0.])

    case(2);          caseName = 'Flows over triangle'
       Lx = 4.0;      Ly = 2.0;     Nbound = 7
       allocate(bounds(Nbound))
       call NewBound(bounds(1 ), [0.0,0.0], [0.0,2.0], 2, [3,2], [0.,0.])
       call NewBound(bounds(2 ), [0.0,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
       call NewBound(bounds(3 ), [4.0,4.0], [2.0,0.0], 1, [2,2], [0.,0.])
       call NewBound(bounds(4 ), [4.0,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

       call NewBound(bounds(5 ), [2.0,2.2], [1.0,1.2], 0, [4,1], [0.,0.])
       call NewBound(bounds(7 ), [2.2,2.2], [1.2,0.8], 0, [4,1], [0.,0.])
       call NewBound(bounds(6 ), [2.2,2.0], [0.8,1.0], 0, [4,1], [0.,0.])


    case(3);          caseName = 'Flows over squares'
       Lx = 4.0;      Ly = 2.0;     Nbound = 28
       allocate(bounds(Nbound))
       call NewBound(bounds(1 ), [0.0,0.0], [0.0,2.0], 2, [3,2], [0.,0.])
       call NewBound(bounds(2 ), [0.0,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
       call NewBound(bounds(3 ), [4.0,4.0], [2.0,0.0], 1, [2,2], [0.,0.])
       call NewBound(bounds(4 ), [4.0,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

       call NewBound(bounds(5 ), [0.9,0.9], [1.2,1.4], 1, [1,1], [0.,0.])
       call NewBound(bounds(6 ), [0.9,1.1], [1.4,1.4], 3, [1,1], [0.,0.])
       call NewBound(bounds(7 ), [1.1,1.1], [1.4,1.2], 2, [1,1], [0.,0.])
       call NewBound(bounds(8 ), [1.1,0.9], [1.2,1.2], 4, [1,1], [0.,0.])

       call NewBound(bounds(9 ), [0.9,0.9], [0.6,0.8], 1, [1,1], [0.,0.])
       call NewBound(bounds(10), [0.9,1.1], [0.8,0.8], 3, [1,1], [0.,0.])
       call NewBound(bounds(11), [1.1,1.1], [0.8,0.6], 2, [1,1], [0.,0.])
       call NewBound(bounds(12), [1.1,0.9], [0.6,0.6], 4, [1,1], [0.,0.])

       call NewBound(bounds(13), [1.9,1.9], [1.4,1.6], 1, [1,1], [0.,0.])
       call NewBound(bounds(14), [1.9,2.1], [1.6,1.6], 3, [1,1], [0.,0.])
       call NewBound(bounds(15), [2.1,2.1], [1.6,1.4], 2, [1,1], [0.,0.])
       call NewBound(bounds(16), [2.1,1.9], [1.4,1.4], 4, [1,1], [0.,0.])
```

```

      call NewBound(bounds(17), [1.9,1.9], [0.4,0.6], 1, [1,1], [0.,0.])
      call NewBound(bounds(18), [1.9,2.1], [0.6,0.6], 3, [1,1], [0.,0.])
      call NewBound(bounds(19), [2.1,2.1], [0.6,0.4], 2, [1,1], [0.,0.])
      call NewBound(bounds(20), [2.1,1.9], [0.4,0.4], 4, [1,1], [0.,0.])

      call NewBound(bounds(21), [2.9,2.9], [1.2,1.4], 1, [1,1], [0.,0.])
      call NewBound(bounds(22), [2.9,3.1], [1.4,1.4], 3, [1,1], [0.,0.])
      call NewBound(bounds(23), [3.1,3.1], [1.4,1.2], 2, [1,1], [0.,0.])
      call NewBound(bounds(24), [3.1,2.9], [1.2,1.2], 4, [1,1], [0.,0.])

      call NewBound(bounds(25), [2.9,2.9], [0.6,0.8], 1, [1,1], [0.,0.])
      call NewBound(bounds(26), [2.9,3.1], [0.8,0.8], 3, [1,1], [0.,0.])
      call NewBound(bounds(27), [3.1,3.1], [0.8,0.6], 2, [1,1], [0.,0.])
      call NewBound(bounds(28), [3.1,2.9], [0.6,0.6], 4, [1,1], [0.,0.])

   case(4);          caseName = 'Driven Cavity'
      Lx = 2.0;     Ly = 2.0;     Nbound = 4
      allocate(bounds(Nbound))
      call NewBound(bounds(1 ), [0.0,0.0], [0.0,2.0], 2, [1,1], [0.,0.])
      call NewBound(bounds(2 ), [0.0,2.0], [2.0,2.0], 4, [1,1], [1.,0.])
      call NewBound(bounds(3 ), [2.0,2.0], [2.0,0.0], 1, [1,1], [0.,0.])
      call NewBound(bounds(4 ), [2.0,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

   case(5);          caseName = 'Contraction channel'
      Lx = 4.0;     Ly = 2.0;     Nbound = 12
      allocate(bounds(Nbound))
      call NewBound(bounds(1 ), [0.0,0.0], [0.0,2.0], 2, [3,1], [0.,0.])
      call NewBound(bounds(2 ), [0.0,1.5], [2.0,2.0], 4, [1,1], [0.,0.])
      call NewBound(bounds(3 ), [1.5,1.5], [2.0,1.5], 1, [1,1], [0.,0.])
      call NewBound(bounds(4 ), [1.5,2.5], [1.5,1.5], 4, [1,1], [0.,0.])
      call NewBound(bounds(5 ), [2.5,2.5], [1.5,2.0], 2, [1,1], [0.,0.])
      call NewBound(bounds(6 ), [2.5,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
      call NewBound(bounds(7 ), [4.0,4.0], [2.0,0.0], 1, [2,2], [0.,0.])
      call NewBound(bounds(8 ), [4.0,2.5], [0.0,0.0], 3, [1,1], [0.,0.])
      call NewBound(bounds(9 ), [2.5,2.5], [0.0,0.5], 2, [1,1], [0.,0.])
      call NewBound(bounds(10), [2.5,1.5], [0.5,0.5], 3, [1,1], [0.,0.])
      call NewBound(bounds(11), [1.5,1.5], [0.5,0.0], 1, [1,1], [0.,0.])
      call NewBound(bounds(12), [1.5,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

   case(6);          caseName = 'Orthogonal channel'
      Lx = 4.0;     Ly = 2.0;     Nbound = 8
      allocate(bounds(Nbound))
      call NewBound(bounds(1 ), [0.0,0.0], [0.0,1.0], 2, [3,1], [0.,0.])
      call NewBound(bounds(2 ), [0.0,1.5], [1.0,1.0], 4, [1,1], [0.,0.])
      call NewBound(bounds(3 ), [1.5,1.5], [1.0,2.0], 2, [1,1], [0.,0.])
      call NewBound(bounds(4 ), [1.5,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
      call NewBound(bounds(5 ), [4.0,4.0], [2.0,1.0], 1, [2,2], [0.,0.])
      call NewBound(bounds(6 ), [4.0,2.5], [1.0,1.0], 3, [1,1], [0.,0.])
      call NewBound(bounds(7 ), [2.5,2.5], [1.0,0.0], 1, [1,1], [0.,0.])
      call NewBound(bounds(8 ), [2.5,0.0], [0.0,0.0], 3, [1,1], [0.,0.])

   case(7);           caseName = 'Suddenly Expanded & square'
      Lx = 4.0;     Ly = 2.0;     Nbound = 12
      allocate(bounds(Nbound))
      call NewBound(bounds(1 ), [0.0,0.0], [0.5,1.5], 2, [3,2], [0.,0.])
      call NewBound(bounds(2 ), [0.0,1.0], [1.5,1.5], 4, [1,1], [0.,0.])
```

```
       call NewBound(bounds(3 ), [1.0,1.0], [1.5,2.0], 2, [1,1], [0.,0.])
       call NewBound(bounds(4 ), [1.0,4.0], [2.0,2.0], 4, [1,1], [0.,0.])
       call NewBound(bounds(5 ), [4.0,4.0], [2.0,0.0], 1, [2,2], [0.,0.])
       call NewBound(bounds(6 ), [4.0,1.0], [0.0,0.0], 3, [1,1], [0.,0.])
       call NewBound(bounds(7 ), [1.0,1.0], [0.0,0.5], 2, [1,1], [0.,0.])
       call NewBound(bounds(8 ), [1.0,0.0], [0.5,0.5], 3, [1,1], [0.,0.])

       call NewBound(bounds(9 ), [2.5,2.5], [0.8,1.2], 1, [1,1], [0.,0.])
       call NewBound(bounds(10), [2.5,3.0], [1.2,1.2], 3, [1,1], [0.,0.])
       call NewBound(bounds(11), [3.0,3.0], [1.2,0.8], 2, [1,1], [0.,0.])
       call NewBound(bounds(12), [3.0,2.5], [0.8,0.8], 4, [1,1], [0.,0.])

    case default;    caseName = 'Default case'
       Lx = 6.0;     Ly = 3.0;      Nbound = 12
       allocate(bounds(Nbound));
       call NewBound(bounds(1 ), [0.0,0.0], [0.5,2.5], 2, [3,1], [0.,0.])
       call NewBound(bounds(2 ), [0.0,3.0], [2.5,2.5], 4, [1,1], [0.,0.])
       call NewBound(bounds(3 ), [3.0,3.0], [2.5,3.0], 2, [1,1], [0.,0.])
       call NewBound(bounds(4 ), [3.0,4.0], [3.0,3.0], 4, [1,1], [0.,0.])
       call NewBound(bounds(5 ), [4.0,4.0], [3.0,2.5], 1, [1,1], [0.,0.])
       call NewBound(bounds(6 ), [4.0,6.0], [2.5,2.5], 4, [1,1], [0.,0.])
       call NewBound(bounds(7 ), [6.0,6.0], [2.5,0.5], 1, [2,2], [0.,0.])
       call NewBound(bounds(8 ), [6.0,2.0], [0.5,0.5], 3, [1,1], [0.,0.])
       call NewBound(bounds(9 ), [2.0,2.0], [0.5,0.0], 1, [1,1], [0.,0.])
       call NewBound(bounds(10), [2.0,1.0], [0.0,0.0], 3, [1,1], [0.,0.])
       call NewBound(bounds(11), [1.0,1.0], [0.0,0.5], 2, [1,1], [0.,0.])
       call NewBound(bounds(12), [1.0,0.0], [0.5,0.5], 3, [1,1], [0.,0.])
 end select
end subroutine geometry
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine InitGrid()
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! Staggered Grid:
!
! I: 1  ... ...  i2 ...  ... i3 .. i4  ....    i5  ...  ... ...  i6    +
!    :           :           :     :           :                 :
!    :           :           :  P  U  P  U  P  U  P              :
!    :           :           :  V ||||V|||||V|||| V..............:.....j4
!    :           :           :  P |U| P  U  P |U| P              :     :
!    :           :           :  V ||| V     V ||| V              :     :
! P  U  P  U  P  U  P  U  P  U  P |U| P     P |U| P  U  P  U  P  U  P  :
! V ||||V|||||V|||||V|||||V|||||V|||| V     V ||||V|||||V|||||V|||| V..j3
! P |U| P  U  P  U  P  U  P  U  P  U  P     P  U  P  U  P  U  P |U| P  :
! V ||| V        :                                            V ||| V  :
! P |U| P        :           ---V---     P: pressure          P |U| P  :
! V ||| V        :          |   |   |                         V ||| V  :
! P |U| P        :          U---P---U    U: dx/dt             P |U| P  :
! V ||| V        :          |   |   |                         V ||| V  :
! P |U| P        :           ---V---     V: dy/dt             P |U| P  :
! V ||| V        :                                            V ||| V  :
! P |U| P  U  P  U  P     P  U  P  U  P  U  P  U  P  U  P  U  P |U| P  :
! V ||||V|||||V|||| V     V ||||V|||||V|||||V|||||V|||||V|||||V|||||V..j2
! P  U  P  U  P |U| P     P |U| P  U  P  U  P  U  P  U  P  U  P  U  P  :
!             V ||| V     V ||| V                                      :
```

```
!             P |U| P  U  P |U| P                                      :
!             V ||||V|||||V|||| V ....................................  1
! +           P  U  P  U  P  U  P                                      J
!
! i = x/dx + 1;   j = y/dy + 1
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 use parameters; use grid; use boundary; use variables
 implicit none

 nx = anint(Lx/dx) + 1;       ny = anint(Ly/dy) + 1;
 allocate(u(nx  , ny+1));     allocate(uStar(nx  , ny+1))
 u = 0.0;                     uStar = 0.0
 allocate(v(nx+1, ny  ));     allocate(vStar(nx+1, ny  ))
 v = 0.0;                     vStar = 0.0
 allocate(p(nx+1, ny+1));     allocate(pCorr(nx+1, ny+1));
 p = 0.0;                     pCorr = 0.0

 allocate(xp(nx+1,ny+1));     allocate(yp(nx+1,ny+1))
 allocate(xu(nx+0,ny+1));     allocate(yu(nx+0,ny+1))
 allocate(xv(nx+1,ny+0));     allocate(yv(nx+1,ny+0))
 do i = 1, nx
    xu(i,:) = (i-1.0)*dx;
 end do
 do j = 1, ny+1
    yu(:,j) = (j-1.5)*dy;
    yp(:,j) = (j-1.5)*dy;
 end do
 do i = 1, nx+1
    xv(i,:) = (i-1.5)*dx;
    xp(i,:) = (i-1.5)*dx;
 end do
 do j = 1, ny
    yv(:,j) = (j-1.0)*dy;
 end do

 allocate(uin(nx  , ny+1));   allocate(uon(nx  , ny+1))
 allocate(vin(nx+1, ny  ));   allocate(von(nx+1, ny  ))
 allocate(pin(nx+1, ny+1));   allocate(pon(nx+1, ny+1))
 call inpoly(xu, yu, nx  , ny+1, uin, uon)
 call inpoly(xv, yv, nx+1, ny  , vin, von)
 call inpoly(xp, yp, nx+1, ny+1, pin, pon)

end subroutine InitGrid
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine boundaryConditions()
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Boundary condition:
!
!                 Top                    !  LeftTop            RightTop
!                                        !          /         \
!            U P U P U P U               !      U P/U/P       P\U\P U
!           ---V---V---V--               !    V  /V/  V       V  \V\  V
!            U P U P U P U               !  U P/U/P U           U P\U\P U
!      V | V               V | V         !   /V/  V               V  \V\
```

```
!      P U P               P U P        !   /                         \
! Left V | V               V | V        !
!      P U P  flow field   P U P  Right !
!      V | V               V | V        !
!      P U P               P U P        !   \                         /
!      V | V               V | V        !  P\U\P U               U P/U/P
!            U P U P U P U              !  V  \V\  V           V  /V/  V
!            --V---V---V--              !    U P\U\P U       U P/U/P U
!            U P U P U P U              !      V  \V\          /V/  V
!                                       !           \          /
!               Bottom                  !  LeftBottom        RightBottom
!
!
!                          Solid boundary conditions
! =======================================================================
! :   :              Top               :             Bottom            :
! :.....................................................................
! : U :      u(i,j+1) = - u(i,j)       :         u(i,j) = - u(i,j-1)   :
! :...:.................................................................
! : V :        V(i,j) =   0            :         V(i,j) =   0          :
! :...:.................................................................
! : P :      p(i,j+1) =   p(i,j)       :         p(i,j) =   p(i,j-1)   :
! =======================================================================
! :   :              Left              :              Right            :
! :.....................................................................
! : U :        u(i,j) =   0            :         u(i,j) = 0            :
! :...:.................................................................
! : V :        v(i,j) = - v(i+1,j)     :       v(i+1,j) = - v(i,j)     :
! :...:.................................................................
! : P :        p(i,j) =   p(i+1,j)     :       p(i+1,j) =   p(i,j)     :
! =======================================================================
! :   :            LeftTop             :            RightTop           :
! :.....................................................................
! : U :        u(i,j) =   0            :         u(i,j) =   0          :
! :...:.................................................................
! : V :        v(i,j) =   0            :         v(i,j) =   0          :
! :...:.................................................................
! : P : p(i,j) = [p(i-1,j)+p(i,j+1)]/2 : p(i,j) = [p(i+1,j)+p(i,j+1)]/2:
! =======================================================================
! :   :           LeftBottom           :           RightBottom         :
! :.....................................................................
! : U :        u(i,j) =   0            :         u(i,j) =   0          :
! :...:.................................................................
! : V :        v(i,j) =   0            :         v(i,j) =   0          :
! :...:.................................................................
! : P : p(i,j) = [p(i-1,j)+p(i,j-1)]/2 : p(i,j) = [p(i+1,j)+p(i,j-1)]/2:
! =======================================================================
!
!                   Input and output boundary conditions
! =======================================================================
! :   :          Input(Left)           :         Output(Right)         :
! :.....................................................................
! : U :       u(i+1,j) = u(i,j)        :       u(i+1,j) = u(i,j)       :
! :...:.................................................................
! : V :       v(i+1,j) = v(i,j)        :       v(i+1,j) = v(i,j)       :
! :...:.................................................................
! : P :         p(i,j) = p(i+1,j)      :       p(i+1,j) = p(i,j)       :
```

```fortran
! ======================================================================
use variables; use boundary
implicit none
real(o)                  :: y                 !
real(o)                  :: ymin,  ymax       !

do i = 1, Nbound
   side  = bounds(i)%side
   imin  = minval(bounds(i)%i);  imax = maxval(bounds(i)%i)
   jmin  = minval(bounds(i)%j);  jmax = maxval(bounds(i)%j)
   typeU = bounds(i)%typeU;      Uval = bounds(i)%Uval
   typeV = bounds(i)%typeV;      Vval = bounds(i)%Vval
   ymin  = minval(bounds(i)%y);  ymax = maxval(bounds(i)%y)
   select case(side)
      case(1) ! right
         ! Boundary condition for u
         if (typeU == 1) then
            u(imin,jmin+1:jmax) = Uval
         else if (typeU == 2) then
            u(imin,jmin+1:jmax) = u(imin-1,jmin+1:jmax)
         end if

         !Boundary condition for v
         if (typeV == 1) then
            v(imin+1,jmin+1:jmax-1) = -v(imin,jmin+1:jmax-1) + 2*Vval
         else if (typeV == 2) then
            v(imin+1,jmin+1:jmax-1) =  v(imin,jmin+1:jmax-1)
         end if

         !p(imin+1, jmin+1:jmax) = p(imin, jmin+1:jmax)
      case(2) ! left
         ! Boundary condition for u
         if (typeU == 1) then
            u(imin,jmin+1:jmax) = Uval;
         else if (typeU == 2) then
            u(imin,jmin+1:jmax) = u(imin+1,jmin+1:jmax)
         else if (typeU == 3) then ! input boundary
            do j = jmin+1, jmax
               y = ymin + (j-jmin)*dy - dy/2
               if (y <= ymin+0.1) then
                  u(1,j) = 10*(y-ymin)
               else if (y >= ymax-0.1) then
                  u(1,j) = 10*(ymax-y)
               else
                  u(1,j) = 1
               end if
            end do
         end if

         !Boundary condition for v
         if (typeV == 1) then
            v(imin,jmin+1:jmax-1) = -v(imin+1,jmin+1:jmax-1) + 2*Vval
         else if (typeV == 2) then
            v(imin+1,jmin+1:jmax-1) =  v(imin,jmin+1:jmax-1)
         end if

         !p(imin, jmin+1:jmax) = p(imin+1, jmin+1:jmax)
```

```fortran
       case(3) ! bottom
           !Boundary condition for u
           if (typeU == 1) then
              u(imin+1:imax-1,jmin) = - u(imin+1:imax-1,jmin+1) + 2*Uval
           else if (typeU == 2) then
              u(imin+1:imax-1,jmin) = u(imin+1:imax-1,jmin+1)
           end if

           !Boundary condition for v
           if (typeV == 1) then
              v(imin+1:imax,jmin) = Vval
           else if (typeV == 2) then
              v(imin+1:imax,jmin) = v(imin+1:imax,jmin+1)
           end if

           !p((imin+1):imax, jmin) = p((imin+1):imax, jmin+1)
       case(4) ! top
           !Boundary condition for u
           if (typeU == 1) then
              u(imin+1:imax-1,jmin+1) = - u(imin+1:imax-1,jmin) + 2*Uval
           else if (typeU == 2) then
              u(imin+1:imax-1,jmin+1) = u(imin+1:imax-1,jmin)
           end if

           !Boundary condition for v
           if (typeV == 1) then
              v(imin+1:imax,jmin) = Vval
           else if (typeV == 2) then
              v(imin+1:imax,jmin) = v(imin+1:imax,jmin-1)
           end if

       case default
           forall(i=imin:imax, j=jmin:jmax, uon(i,j))
             u(i,j) = 0
           end forall
           forall(i=imin:imax, j=jmin:jmax, von(i,j))
             v(i,j) = 0
           end forall
    end select

 end do

end subroutine boundaryConditions
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

subroutine corner()
! Deal with the pressure value of the corners for output
!
! Corner:
!
!        TL    P U P       P U P   TR      TL: Top & Left
!           \  V | V       V | V   /           p(i,j) = [ p(i+1,j   )
!            \ P U P       P U P  /                      + p(i   ,j-1)
!             \V | V       V | V/                        + p(i+1,j-1)]/3
!      P U P U P U P       P U P U P U P   TR: Top & Right
!     -V---V---V-+ V       V +-V---V---V-      p(i,j) = [ p(i-1,j   )
```

```
!      P U P U P U P       P U P U P U P                  + p(i  ,j-1)
!                                                         + P(i-1,j-1)]/3
!                flow field
!                                              BL: Bottom & Left
!      P U P U P U P       P U P U P U P           p(i,j) = [  p(i  ,j+1)
!     -V---V---V-+ V       V +-V---V---V-                 + p(i+1,j  )
!      P U P U P U P       P U P U P U P                  + p(i+1,j+1)]/3
!             /V | V       V | V\              BR: Bottom & Right
!            / P U P       P U P \                 p(i,j) = [  p(i-1,j  )
!           /  V | V       V | V  \                       + p(i  ,j+1)
!         BL   P U P       P U P   BR                     + p(i-1,j+1)]/3
!
 use variables; use boundary

 integer                 :: ic1, jc1, ic2, jc2
 integer                 :: iFirst, jFirst, boundFirst
 integer                 :: cornerType = 0

 boundFirst = 1
 iFirst = bounds(boundFirst)%i(1); jFirst = bounds(boundFirst)%j(1)

 do i = 1, Nbound
    cornerType = 0
    ic1 = bounds(i)%i(2); jc1 = bounds(i)%j(2)
        if (i<Nbound)  ic2 = bounds(i+1)%i(1);jc2 = bounds(i+1)%j(1)
    if ((ic1 == ic2).and.(jc1 == jc2).and.i<Nbound) then
       if (bounds(i)%side*bounds(i+1)%side==4) cornerType = 1
       if (bounds(i)%side*bounds(i+1)%side==3) cornerType = 2
       if (bounds(i)%side*bounds(i+1)%side==6) cornerType = 3
       if (bounds(i)%side*bounds(i+1)%side==8) cornerType = 4
    else if((ic1 == iFirst).and.(jc1 == jFirst)) then
       if (bounds(i)%side*bounds(boundFirst)%side==4) cornerType = 1
       if (bounds(i)%side*bounds(boundFirst)%side==3) cornerType = 2
       if (bounds(i)%side*bounds(boundFirst)%side==6) cornerType = 3
       if (bounds(i)%side*bounds(boundFirst)%side==8) cornerType = 4
       if (i < Nbound) then
           boundFirst = i+1
           iFirst = bounds(boundFirst)%i(1)
           jFirst = bounds(boundFirst)%j(1)
           end if
    end if

    select case(cornerType)
       case(1)
           ic1 = ic1+1; jc1 = jc1+1;
           p(ic1,jc1) = (p(ic1,jc1-1)+p(ic1-1,jc1)) /2
       case(2)
           ic1 = ic1+1; jc1 = jc1;
           p(ic1,jc1) = (p(ic1-1,jc1)+p(ic1,jc1+1)) /2
       case(3)
           ic1 = ic1; jc1 = jc1;
           p(ic1,jc1) = (p(ic1+1,jc1)+p(ic1,jc1+1)) /2
       case(4)
           ic1 = ic1; jc1 = jc1+1;
           p(ic1,jc1) = (p(ic1+1,jc1)+p(ic1,jc1-1)) /2
    end select
 end do
```

```fortran
end subroutine corner



!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine explicitEuler()
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Primitive variables formulation of NS equations:
!                                                   _              _
!            du    dp      d(uu)   d(uv)    1  |  d^2u     d^2u   |
!            -- + -- = - ----- - ----- + -- |  ---- + ----   |
!            dt    dx       dx       dy      Re |_  dx^2    dy^2 _|
!
!                                                   _              _
!            dv    dp      d(uv)   d(vv)    1  |  d^2v     d^2v   |
!            -- + -- = - ----- - ----- + -- |  ---- + ----   |
!            dt    dy       dx       dy      Re |_  dx^2    dy^2 _|
!
!
!            du    dv
!            -- + -- = 0
!            dx    dy
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Discretization Schemes:
!
!             n+1     n          /             1               \
!            U     = U     +  dt*| - A - B + --- ( C + D ) |
!             ij      ij          \            Re              /
!
!
!             u(i+1,j) - u (i-1,j)             u(i,j+1) - u(i,j-1)
! A = u(i,j) --------------------    ,  B = vij ------------------
!                     2dx                                2dy
!
!                                          v(i,j-1)+v(i+1,j-1)+v(i,j)+v(i+1,j)
!                                   vij = -----------------------------------
!                                                          4
!
!      u(i+1,j) - 2u(i,j) + u(i-1,j)          u(i,j+1) - 2u(i,j) + u(i,j-1)
! C = ----------------------------- ,  D = -----------------------------
!                 dx*dx                                 dy*dy
!
!::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
!
!             n+1     n          /             1                    \
!            v     = v     +  dt*| - E - F + --- ( G + H )  - g |,      g = 0
!             ij      ij          \            Re                   /
!
!
!             v(i+1,j) - v (i-1,j)            v(i+1,j) - v(i-1,j)
! E = v(i,j) --------------------    ,  F = vij -------------------
!                   2dy                                2dx
!
!                                          v(i,j-1)+v(i+1,j-1)+v(i,j)+v(i+1,j)
!                                   vij = -----------------------------------
!                                                         4
```

```
!
!     v(i+1,j) - 2v(i,j) + v(i-1,j)          v(i,j+1) - 2v(i,j) + v(i,j-1)
! G = ----------------------------- ,  H = -----------------------------
!                 dx*dx                                dy*dy
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 use variables; use boundary; use grid
 implicit none
 uStar = u
 forall(i=2:nx-1, j=2:ny, (uin(i,j).and.(.not.uon(i,j))))
    uStar(i,j) = u(i,j) - dt/dx*(p(i+1,j)-p(i,j))                    &
                 + dt*(                                               &
                       - u(i,j)*(u(i+1,j) - u(i-1,j))/(2*dx)          &!A
                       - (v(i,j-1)+v(i+1,j-1)+v(i,j)+v(i+1,j))/4.0    &!B
                        *(u(i,j+1) - u(i,j-1))/(2*dy)                 &
                       + (    (u(i+1,j) - 2*u(i,j) + u(i-1,j))/dx**2  &!C
                             +(u(i,j+1) - 2*u(i,j) + u(i,j-1))/dy**2  &!D
                                                                  )/Re)
 end forall

 vStar = v
 forall(i=2:nx, j=2:ny-1, (vin(i,j).and.(.not.von(i,j))))
    vStar(i,j) = v(i,j) - dt/dy*(p(i,j+1)-p(i,j))                    &
                 + dt*(                                               &
                        - (u(i,j)+u(i,j+1)+u(i-1,j)+u(i-1,j+1))/4.0   &!E
                         *(v(i+1,j) - v(i-1,j))/(2*dx)                &
                        - v(i,j)*(v(i,j+1) - v(i,j-1))/(2*dy)         &!F
                        + (  (v(i+1,j)- 2*v(i,j)+ v(i-1,j))/dx**2     &!G
                            +(v(i,j+1)- 2*v(i,j)+ v(i,j-1))/dy**2     &!H
                                                                 )/Re)
 end forall

end subroutine explicitEuler
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine presProj()
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Use simple 4 points scheme for Laplace operator.For one iterative step
! of poisson equation solver can be written as follows:
!
!          p'(i,j) = [  b(  p'(i+1,j) + p'(i-1,j)  )
!                         + c(  p'(i,j+1) + p'(i,j-1)  ) + d ] / a
! where:
!          b = dt/dx/dx,  c = dt/dy/dy,  a = 2(b+c)
!
!          d = [ u(i+1,j) - u(i-1,j) ]/dx + [ u(i,j+1) - u(i,j+1) ]/dy
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 use parameters; use variables; use boundary; use grid
 implicit none
 real(o), parameter      :: b = dt/dx/dx      !
 real(o), parameter      :: c = dt/dy/dy      !
 real(o)                 :: a = 2*b + 2*c     !
 real(o)                 :: alpha = 0.5       ! coefficient of relaxation
 real(o)                 :: dp                ! pressure error
 real(o)                 :: p0(nx+1, ny+1)    ! pressure temp
```

```
 real(o)                :: res(nx+1,ny+1)        ! velocity dispersion


 res = 0
 forall(i = 2:nx, j = 2:ny, pin(i,j))        ! res = du/dx + dv/dy
    res(i,j) =    (uStar(i,j)-uStar(i-1,j))/dx                         &
                + (vStar(i,j)-vStar(i,j-1))/dy
 end forall
 resmax = maxval(abs(res))

 p0 = 0;  pCorr = 0; dp = 2*prec
 ! solve pressure-correction equation
 do while(dp>prec)
    forall(i=2:nx, j=2:ny,pin(i,j))
       pCorr(i,j)= 1/a*(b*p0(i-1,j) + b*p0(i+1,j) +                    &
                        c*p0(i,j-1) + c*p0(i,j+1) -res(i,j))
    end forall
    dp = maxval(abs(pCorr-p0))
    p0 = pCorr
 end do

 ! corrected values of velocity fields
 forall(i = 1:nx, j = 1:ny,uin(i,j))
    u(i,j) = uStar(i,j) - dt/dx*(pCorr(i+1,j) - pCorr(i,j))
 end forall
 forall(i = 1:nx, j = 1:ny,vin(i,j))
    v(i,j) = vStar(i,j) - dt/dy*(pCorr(i,j+1) - pCorr(i,j))
 end forall

 ! corrected values of pressure fields
 forall(i = 2:nx, j = 2:ny,pin(i,j))
    p(i,j) = p(i,j) + alpha*pCorr(i,j)
 end forall

end subroutine presProj
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine output()
!  Write output data file
!
!  Output:  Data file "solution.dat" containing for each point
!           the following:
!           coordinate, velocity, pressure

 use variables; use parameters; use grid; use boundary; implicit none
 real(o)                :: xNode(nx, ny), yNode(nx, ny)
 real(o)                :: uNode(nx, ny)
 real(o)                :: vNode(nx, ny)
 real(o)                :: pNode(nx, ny)
 real(o)                :: ptemp(nx, ny)
 logical                :: NodeIn(nx,ny), NodeOn(nx,ny)


 call boundaryConditionsp(p)
 call corner()
 xNode = xu(:,1:ny);    yNode = yv(1:nx,:)
```

```fortran
  call inpoly(xNode, yNode, nx, ny, NodeIn, NodeOn)
  pNode = 0; uNode = 0; vNode = 0;

  ! Node values are explicitly obtained by averaging the cell data.
  forall(i=1:nx, j=1:ny, NodeIn(i,j))
     uNode(i,j) = ( u(i,j) + u(i,j+1) )/2
     vNode(i,j) = ( v(i,j) + v(i+1,j) )/2
     pNode(i,j) = (p(i,j)+p(i+1,j)+p(i,j+1)+p(i+1,j+1))/4
  end forall

  ! the volecity of the boundarys are set to be zeros
  do k = 1, Nbound
     imin  = minval(bounds(k)%i);  imax = maxval(bounds(k)%i)
     jmin  = minval(bounds(k)%j);  jmax = maxval(bounds(k)%j)
     typeU = bounds(k)%typeU;      Uval = bounds(k)%Uval
     typeV = bounds(k)%typeV;      Vval = bounds(k)%Vval
     if (typeU == 1.or.typeU == 0)  then
        forall(i=imin:imax, j=jmin:jmax, NodeOn(i,j)) uNode(i,j) = Uval
     end if
     if (typev == 1.or.typev == 0)  then
        forall(i=imin:imax, j=jmin:jmax, NodeOn(i,j)) vNode(i,j) = Vval
     end if
  end do

  open(unit = 1, file='solution.dat', status = 'replace')
  write(1,*), '% Solution to 2D Incompressible NS Equations with SIMPLE %'
  write(1,*), '% computational case :', caseName
  write(1,*), '%'
  write(1,*), '% % % % % % % % % % % % parameters % % % % % % % % % % % %'
  write(1, '(" %  Reynolds number   :", f7.2)') Re
  write(1, '(" %  spatial step size :", f5.2)') dx
  write(1, '(" %  time step         :", f7.4)') dt
  write(1,*), '%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%'
  write(1,*)
  write(1, '(" %", 6x, "x", 9x, "y", 9x, "u", 9x, "v", 9x, "p")')
  do i = 1, nx; do j = 1, ny
      if (NodeIn(i,j)) then
         write(1,'(5f10.4)') xNode(i,j), yNode(i,j),                  &
                             uNode(i,j), vNode(i,j), pNode(i,j)
      end if
  end do; end do
  close(1)
  write(*,*),  "Done! The flow data are writen to 'solution.dat' "

end subroutine output
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine inpoly(x, y, r, c, cn, on)
! Determine whether a series of points lie within the bounds of a polygon
! in the 2D plane. General non-convex, multiply-connected polygonal
! regions can be handled.
!
! The algorithm is based on the crossing number test,   which counts the
! number of times a line that extends from each point past the right-most
! region of the polygon intersects with a polygon edge.   Points with odd
! counts are inside.
```

```
! http://local.wasp.uwa.edu.au/~pbourke/geometry/insidepoly/
!
!   x,y           : The points to be tested.
!   xbound,ybound: An Mx2 array of polygon edges,specified as connections
!        between the vertices in NODE: [x1 x2; x3 x4; etc].  The vertices
!        in NODE do not need to  be specified in connsecutive  order when
!        using the extended syntax.
!
!   in   : An logical array with  IN(i,j) = TRUE if  [x(i,j),y(i,j)]  lies
!          within the region.
!   on   : An logical array with  ON(i,j) = TRUE if  [x(i,j),y(i,j)]  lies
!          on a polygon edge.  (A tolerance is used to deal with numerical
!          precision, so that points within a distance of less than 10e-12
!          from a polygon edge are considered "on" the edge.

 use variables; use boundary
 implicit none
 integer, intent(in)    :: r, c              ! size of test points array
 real(o), intent(in)    :: x(r,c), y(r,c)    ! test points array
 real(o)                :: tol = 1.0e-6      ! tolerance
 logical, intent(out)   :: cn(r,c)           ! logical array
 logical, intent(out)   :: on(r,c)           ! logical array
 real(o)                :: x1, x2, y1, y2    ! edge: (x1,y1), (x2,y2)
 real(o)                :: xmin, xmax        ! xmin = min(x1,x2) ...
 real(o)                :: xi, yi            ! (xi,yi) = (x(i,j), y(i,j))

 on = .false. ; cn = .false.

 do k = 1, Nbound          ! Loop through edges
   ! Nodes in current edge
   y1 = bounds(k)%y(1); y2 = bounds(k)%y(2)
   if (y1<y2) then
      x1 = bounds(k)%x(1); x2 = bounds(k)%x(2)
   else
      x1 = y1; y1 = y2; y2 = x1;
      x1 = bounds(k)%x(2); x2 = bounds(k)%x(1)
   end if
   xmin = min(x1,x2);   xmax = max(x1,x2);
   ! Loop through points
   do i = 1,r
       do j = 1,c
          ! Check the bounding-box for the edge before doing the
          ! intersection test. Take shortcuts wherever possible!
           yi = y(i,j); xi = x(i,j)
           if ( y1<=yi+tol .and. yi<=y2+tol ) then
               if ( xi+tol>=xmin .and. xi<xmax+tol ) then
                  ! Check if we're "on" the edge
                  on(i,j) = on(i,j).or.                                     &
                            (abs((y2-yi)*(x1-xi)-(y1-yi)*(x2-xi))<tol)
               end if
           end if

           if ( (y1<=yi) .and. (yi<=y2) ) then
               if (xi>=xmin) then
                   if (xi<=xmax) then
                       ! Do the actual intersection test
                       if ((y2-y1)*(xi-x1)<(yi-y1)*(x2-x1))then
```

```
                          cn(i,j) = (.not.cn(i,j))
                       end if
                   end if
               elseif (yi<y2) then ! Deal with points exactly at vertices
                   cn(i,j) = (.not.cn(i,j))
               end if
           end if
       end do
   end do
 end do
! Re-index to undo the sorting
 forall(i=1:r, j=1:c)
    cn(i,j) = (cn(i,j) .or. on(i,j))
 end forall

end subroutine
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine progressBar()
! draw the program running process bar
!    resmax(t0) * reduceFactor^nbarmax <= toleror
!    reduceFactor = (toleror/resmax(t0))**(1/nbarmax)
!
!    reduceFactor = (toleror/resmax(t))**(1/nbar)
!    nbar = log(toleror/resmax)/log(reduceFactor)
 use parameters;use process; use variables;
 implicit none
 integer :: resmaxinteger
 character(len=67)::bar
 bar="         |                                                  |?????"
  if (resmax0==0) then
     write(*,*)
     write(*,*),'Running process:'
     write(*,*),                                                       &
       '  max D |0%------20%-------40%-------60%-------80%-----100%|time'
     resmax0 = resmax; redfac = (toleror/resmax)**(1.0/nbarmax)
  elseif (anint( log(resmax/resmax0)/log(redfac) ) <-10) then
     write(*,*)
     write(*,*) 'We may have some problems, please adjust parameters !'
     stop
  end if
  nbar = anint(log(resmax/resmax0)/log(redfac))
  write(unit=bar(3:9),fmt="(f7.4)")  abs(resmax-toleror)

  do k=1, nbar
     bar(10+k:10+k)="="
  end do
  write(unit=bar(62:67),fmt="(f6.3)") t
  ! print the progress bar.
  write(unit=6,fmt="(a1,a67,$)") char(13), bar
  if (resmax<toleror) write(*,*)
end subroutine progressBar

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine boundaryConditionsp(p)
! Deal with the pressure value of the boundarys for output
```

```
 use grid; use boundary; use parameters;
 implicit none
 real(o), intent(inout) :: p(nx+1, ny+1)
 integer :: i
 do i = 1, Nbound
    side  = bounds(i)%side
    imin  = minval(bounds(i)%i);  imax = maxval(bounds(i)%i)
    jmin  = minval(bounds(i)%j);  jmax = maxval(bounds(i)%j)
    select case(side)
       case(1); p(imin+1, jmin+1:jmax) = p(imin,   jmin+1:jmax)
       case(2); p(imin   , jmin+1:jmax) = p(imin+1, jmin+1:jmax)
       case(3); p(imin+1:imax, jmin   ) = p(imin+1:imax, jmin+1)
       case(4); p(imin+1:imax, jmin+1) = p(imin+1:imax,   jmin)
    end select
 end do
end subroutine boundaryConditionsp
```

B data2figure.m

```
% Get the data of the solution of 2D Incompressible NS Equations with SIMPLE
% and show as a figure. the data: "solution.dat" = Data file containing for
% each grid point, coordinate, velocity and pressure, in the following format:
%
%              x(1)  y(1)     u(1)  v(1)     p(1)
%              x(2)  y(2)     u(2)  v(2)     p(2)
%               .     .        .     .        .
%               .     .        .     .        .
%              x(n)  y(n)     u(n)  v(n)     p(n)
%
% Copyrhigt by Zhou Lvwen: zhou.lv.wen[at]gmail.com

dx = 0.02; dy = 0.02;
solution = load('solution.dat');
x = solution(:,1);
y = solution(:,2);
u = solution(:,3);
v = solution(:,4);
p = solution(:,5);
I = round(x/dx)+1;
J = round(y/dy)+1;

imin=min(I); imax = max(I);
jmin=min(J); jmax = max(J);

U =  zeros(jmax-jmin+1, imax-imin+1);
V =  zeros(jmax-jmin+1, imax-imin+1);
P =  inf*ones(jmax-jmin+1, imax-imin+1);

for k = 1:length(x)
    i = I(k); j = J(k);
    U(j,i) = u(k); V(j,i) = v(k); P(j,i) = p(k);
end

U =  flipud(U); V = -flipud(V); P =  flipud(P);
[X,Y] = meshgrid([[imin:imax]-1]*dx, [[jmin:jmax]-1]*dy);
imagesc(X(:),Y(:),P)
hold on
streamslice(X,Y,U,V,10,'b');
```

```
xlabel('x');ylabel('y')
axis image
```