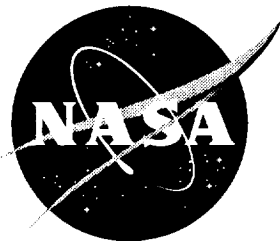


NASA Technical Memorandum 112874



Viscous Driven-Cavity Solver *User's Manual*

William A. Wood
Langley Research Center, Hampton, Virginia

September 1997

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001



Viscous Driven-Cavity Solver

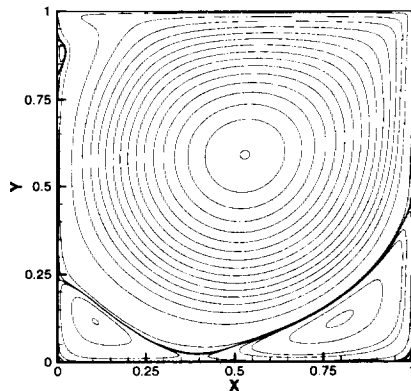
User's Manual

William A. Wood*
NASA Langley Research Center

September 29, 1997

Abstract

The viscous driven cavity problem is solved using a stream-function and vorticity formulation for the incompressible Navier-Stokes equations. Analytical development and results are presented by Wood[1]. Convergence is accelerated by employing grid sequencing and a V-cycle multigrid relaxation. The formulation is second-order accurate in space and first-order implicit in time. The coefficients are lagged to loosely couple the stream-function and vorticity equations. The linearized system is relaxed with symmetric Gauß-Seidel.



* Aerospace Technologist, Aerothermodynamics Branch, Aero- and Gas-Dynamics Division.
w.a.wood@larc.nasa.gov (757) 864-8355.

Contents

Abstract	1
Introduction	3
Params.inc	4
Cavity.inp	5
Main Driver	6
Mesh Sequencing	7
Relaxation on Fine Mesh	8
Inputs	9
Grid	10
dxdy	12
Initial	14
Timestep	15
Solvr	17
Load A	23
Load b	25
Sgs	28
Boundary Conditions	30
Decp	34
Stopping	36
Restrict	38
Prolong	39
Multig	41
Outsol	44
Results	46
Summary	48

Introduction

The present document is the FORTRAN source code for solving the incompressible Navier-Stokes equations inside a two-dimensional driven cavity. The governing equations are cast in a stream-function/vorticity formulation as presented by Wood[1].

Results presented in Ref. [1] focus on streamline patterns, stability issues, and multigrid performance. Additional results presented here provide numerical tabulations and comparisons with results from literature.

The source code is in three parts. The file `params.inc` contains the grid dimensions while the file `cavity.inp` contains the input deck. The file `cavity.f` is the bulk of the code. To execute, compile `cavity.f` and run. This visual User's Manual is derived from the source code *via* processing with the L^AT_EX₂_ε software.

Following the Introduction are the two files which are typically edited between solutions, `params.inc` and `cavity.inp`. Then follows the driver routine and the subroutines, each as a separate section of the manual.

Params.inc

The parameter file, `params.inc`, for cavity sets the number of points in i and j as `imax` and `jmax`.

```
c    parameter file for cavity ... .. params.inc

    parameter ( imax = 257, jmax = 257 )
    parameter ( imxmed = (imax+1)/2, jmxmed = (jmax+1)/2 )
    parameter ( imxcor = (imxmed+1)/2, jmxcor = (jmxmed+1)/2 )

    dimension dx(imax, jmax, 2), x(imax, jmax, 2),
$      f(imax, jmax, 5),
$      alam(imax, jmax, 2), sig(imax, jmax, 2)

    dimension fmed(imax, jmax, 5), fcor(imax, jmax, 5)

    dimension aij(imax, jmax), aip1(imax, jmax), aim1(imax, jmax),
$      ajp1(imax, jmax), ajm1(imax, jmax), b(imax, jmax),
$      res(imax, jmax, 5, 3)

    common / vars / ntot, dx, x, f, alam, sig, fmed, fcor
    common / vecs / aij, aip1, aim1, ajp1, ajm1, b, res
```

The dependent variables are stored in $f(i, j, var)$ where var takes on values of 1-5 corresponding to (u, v, ζ, P, ψ) .

Cavity.inp

The input deck is cavity.inp.

```
input deck to cavity ..... cavity.inp
$grids
alength = 1.
height = 1.
igrd = 0
beta = 1.4
$
$conditions
alammax = 1.
sigmax = 1.
re = 2000.
ubig = 1.
igov = 2
nitsgs = 1
niterp = 1000
convg = 5.e-5
iskpout = 50
img = 2
$

c alength - cavity length
c height - cavity height
c igrd - 0 = uniform grid
c      1 - beta clustering to edges
c beta - clustering strength > 1
c alammax - CFL-like number for inviscid terms
c sigmax - CFL-like number for viscous terms
c re - Reynolds number = ubig * alength / nu
c ubig - upper boundary speed
c igov - 1 = primitive variables formulation
c      2 = stream function/vorticity formulation
c nitsgs - number of sgs sub-sweeps
c niterp - maximum number of iterations
c convg - convergence tolerance
c iskpout - convergence history output skip
c img - 0 = fine mesh only
c      1 = mesh sequencing
c      2 = mesh sequencing + V-cycle multigrid
```

Main Driver

The driver routine contains the logic for mesh sequencing and if multigrid is desired. Subroutines called: `inputs`, `grid`, `dxdy`, `initial`, `timestep`, `solvr`, `restrict`, `bc`, `prolong`, `multig`, `outsol`. The subroutines follow in that order (the order in which they are called).

This code was originally computer assignment 1 for Applied Science 778 at William & Mary, 1996.

```
program cavity
```

Load dimension, common, and parameter statements.

```
include 'params.inc'
logical istop

common / parms / alength, alamax, sigmax, re, height, ubig,
$   igov, nitsgs, niterp, convg, iskpout, img, istop,
$   igrd, beta
common / parm2 / cpustart, dxfin, dxmed, dxcor, iskpfin,
$   iskpmed, iskpcor, resid0, nout
```

Default initial values.

```
istop   = .false.      ! Stopping flag
dxfin   = 1.           ! Fine grid 1:1 spacing, relative to fine
dxmed   = .5           ! Medium grid 1:2 spacing, relative to fine
dxcor   = .25          ! Coarse grid 1:4 spacing, relative to fine
iskpfin = 1
iskpmed = 2            ! Medium grid coarsed by 2 relative to fine
iskpcor = 4            ! Coarse grid coarsed by 4 relative to fine
ntot    = 0
```

Load input deck, `cavity.inp`.

```
call inputs
```

Set CPU timer at start of main routine. `etime` is timer for SUN and SGI. `atime1`, `atime2` are dummy variables.

```
cpustart = etime( atime1, atime2 )
```

Generate the fine grid.

```
imx1 = imax
jmx1 = jmax
call grid ( igrd, imx1, jmx1, alength, height, beta )
```

Define dx and dy grid spacings on finest grid.

```
call dxdy ( imx1, jmx1 )
```


Initialize flowfield on finest grid.

```
call initial ( imx1, jmx1, ubig, dxfin, iskpfin, f(1,1,1) )
```

Set dt - time step size, and lambda, sigma coefficients.

```
call timestep ( alamax, sigmax, re )
```

Run 1 timestep on fine mesh to get starting residual--resid0.

```
resid0 = -1.  
call solvr ( 1, 0, imx1, jmx1, igov, dxfin, iskpfin,  
$ nitsgs, re, ubig, resid0, istop, convg, nout,  
$ iskpout, cpustart, f(1,1,1) )
```

Skip mesh sequencing if img=0. Perform mesh sequencing if img=1 or 2.

```
if ( img .eq. 0 ) go to 100
```

Mesh Sequencing

Restrict from fine to medium mesh.

```
call restrict ( imxmed, jmxmed, f(1,1,1), fmed(1,1,1) )
```

Restrict from medium to coarse mesh.

```
call restrict ( imxcor, jmxcor, fmed(1,1,1), fcor(1,1,1) )
```

Impose boundary conditions.

```
call bc ( igov, imxcor, jmxcor, ubig, re, iskpcor, dxcor,  
$ fcor(1,1,1) )
```

Iterate on coarse mesh for nitc iterations. Options for the number of iterations to perform are,

$$Nit_c = \frac{Imax_c \cdot Jmax_c}{2}$$

$$Nit_c = \max(Imax_c, Jmax_c)$$

$$Nit_c = Imax_c + Jmax_c$$

```
nitc = imxcor * jmxcor / 2  
c nitc = max( imxcor, jmxcor )  
c nitc = imxcor + jmxcor
```

Limit the number of iterations on the coarse mesh to be no more than twice the maximum number of fine mesh iterations.

```
nitc = min( nitc, nitc + nitc )  
call solvr ( nitc, 0, imxcor, jmxcor, igov, dxcor, iskpcor,  
$ nitsgs, re, ubig, resid0, istop, convg, nout,  
$ iskpout, cpustart, fcor(1,1,1) )
```

Prolongate from coarse to medium mesh, impose boundary conditions, and relax on medium mesh.

```
      call prolong ( imxcor, jmxcor, fmed(1,1,1), fcor(1,1,1) )
      call bc ( igov, imxmed, jmxmed, ubig, re, iskpmed, dxmed,
$       fmed(1,1,1) )
      nitc = imxmed * jmxmed / 2
c       nitc = max( imxmed, jmxmed )
c       nitc = imxmed + jmxmed
      nitc = min( nitc, niterp + niterp )
      call solvr ( nitc, 0, imxmed, jmxmed, igov, dxmed, iskpmed,
$       nitsgs, re, ubig, resid0, istop, convg, nout,
$       iskpout, cpustart, fmed(1,1,1) )
```

Now prolongate from medium to fine mesh and impose boundary conditions. Ready to relax on fine mesh.

```
      call prolong ( imxmed, jmxmed, f(1,1,1), fmed(1,1,1) )
      call bc ( igov, imax, jmax, ubig, re, iskpfin, dxfin,
$       f(1,1,1) )
```

Skip point for bypassing mesh sequencing.

```
100 continue
```

Relaxation on Fine Mesh

Is multigrid desired? If so, `img=2`, and we go to subroutine `multig` to converge solution. Otherwise, we relax on the fine grid only.

```
      if ( img .eq. 2 ) then
          call multig
      else
          call solvr ( niterp, 0, imax, jmax, igov, dxfin, iskpfin,
$       nitsgs, re, ubig, resid0, istop, convg, nout,
$       iskpout, cpustart, f(1,1,1) )
      end if
```

Output solution and end of program.

```
      call outsol ( imx1, jmx1, nout )
      stop
      end
```

Inputs

Inputs reads the input deck `cavity.inp` using the `namelist` construct. It is called from `cavity`.

```
subroutine inputs ! ..... inputs

common / parms / alength, alamax, sigmax, re, height, ubig,
$   igov, nitsgs, niterp, convg, iskpout, img, istop,
$   igrd, beta

logical istop

namelist / grids / alength, height, igrd, beta
namelist / conditions / alamax, sigmax, re, ubig, igov,
$   nitsgs, niterp, convg, iskpout, img

open ( 9, file = 'cavity.inp', form = 'formatted' )
read ( 9, * )
read ( 9, grids )
read ( 9, conditions )

return
end
```

Grid

Grid is a 2-D grid generator for rectangular domains. The lower-left corner is set at the origin. Called by `cavity`.

`igrid = 0` — uniform spacing in x and y

`igrid = 1` — Clustering to all walls, of strength $\beta > 1$, with stronger clustering as $\beta \rightarrow 1$.

```
subroutine grid ( igrid, imx, jmx, width, height, beta ) !..grid
```

```
include 'params.inc'
```

Δx and Δy for uniform spacing.

```
dx1 = 1. / float( imx - 1 )
```

```
dy1 = 1. / float( jmx - 1 )
```

$\beta + 1$ and $\beta - 1$.

```
bp1 = beta + 1.
```

```
bm1 = beta - 1.
```

```
d = .5
```

Set $x = 0$ on the left-hand boundary and $x = \text{width}$ on the right-hand boundary.

$x(i, j, 1) = x$ -coordinate, $x(i, j, 2) = y$ -coordinate.

```
do j = 1, jmx
```

```
  x(1, j, 1) = 0.
```

```
  x(imx, j, 1) = width
```

```
end do
```

Set $y = 0$ on the lower boundary and $y = \text{height}$ on the upper boundary.

```
do i = 1, imx
```

```
  x(i, 1, 2) = 0.
```

```
  x(i, jmx, 2) = height
```

```
end do
```

Assign x values between left ($i = 1$) and right ($i = \text{imx}$) sides. x_1 is the arclength fraction of the distance from the left to right for the current point.

```
do i = 2, imx-1
```

Either uniform spacing,

$$x_1 = \Delta x(i-1) = \frac{i-1}{i_{max}-1}$$

```
x1 = dx1 * float( i - 1 )
```

or beta clustering.

$$x_1 = \frac{T + R(\beta - 1) - (\beta + 1)}{T}$$

where,

$$T = (2d + 1)(R + 1), \quad R = \left(\frac{\beta + 1}{\beta - 1} \right)^{\left(\frac{A-d}{1-d} \right)}, \quad A = \frac{i - 1}{i_{max} - 1}$$

```

if ( igrd .eq. 1 ) then
  an = x1
  rn = ( bp1 / bm1 )**((an - d) / (1. - d))
  tn = ( 2. * d + 1. ) * ( rn + 1. )
  x1 = ( tn + rn * bm1 - bp1 ) / tn
end if

do j = 1, jmx
  x(i,j,1) = x(1,1,1) + x1 * width
end do
end do

```

Fill in the y values between bottom ($j = 1$) and top ($j = jmx$) using the same spacing variation as in the x direction.

```

do j = 2, jmx-1

  y1 = dy1 * float( j - 1 )

  if ( igrd .eq. 1 ) then
    an = y1
    rn = ( bp1 / bm1 )**((an - d) / (1. - d))
    tn = ( 2. * d + 1. ) * ( rn + 1. )
    y1 = ( tn + rn * bm1 - bp1 ) / tn
  end if

  do i = 1, imx
    x(i,j,2) = x(1,1,2) + y1 * height
  end do
end do

return
end

```

dxdy

Computes Δx and Δy grid spacings on a cartesian-aligned rectangular grid. Second-order central differencing used for interior points. Second-order one-sided differencing used for boundary points. Called by `cavity`.

```
subroutine dxdy ( imx, jmx ) ! ..... dxdy
```

```
include 'params.inc'
```

At interior points $\Delta x = \frac{1}{2}(x_{i+1} - x_{i-1})$

```
do i = 2, imx - 1
  dx1 = ( x(i+1,1,1) - x(i-1,1,1) ) * .5
  do j = 1, jmx
    dx(i,j,1) = dx1
  end do
end do
```

At the left end $\Delta x = \frac{1}{2}(-3x_1 + 4x_2 - x_3)$

```
dx1 = ( -3. * x(1,1,1) + 4. * x(2,1,1) - x(3,1,1) ) * .5
do j = 1, jmx
  dx(1,j,1) = dx1
end do
```

At the right end $\Delta x = \frac{1}{2}(3x_{imx} - 4x_{imx-1} + x_{imx-2})$

```
dx1 = ( 3.* x(imx,1,1) - 4.* x(imx-1,1,1) + x(imx-2,1,1) ) * .5
do j = 1, jmx
  dx(imx,j,1) = dx1
end do
```

Now do the same for Δy , interior, bottom, and top points.

```
do j = 2, jmx - 1
  dy1 = ( x(1,j+1,2) - x(1,j-1,2) ) * .5
  do i = 1, imx
    dx(i,j,2) = dy1
  end do
end do
```

Bottom.

```
dy1 = ( -3. * x(1,1,2) + 4. * x(1,2,2) - x(1,3,2) ) * .5
do i = 1, imx
  dx(i,1,2) = dy1
end do
```

Top.

```
dy1 = ( 3.* x(1,jmx,2) - 4.* x(1,jmx-1,2) + x(1,jmx-2,2) ) * .5
do i = 1, imx
  dx(i,jmx,2) = dy1
end do

return
end
```

Initial

Sets initial conditions for zero motion at $t = 0^-$. At $t = 0^+$ the upper surface impulsively started at speed U . Called by `cavity`.

```
subroutine initial ( imx, jmx, u, dxfac, iskp, tf ) !...initial
include 'params.inc'
```

`tf` is the array of solution variables to be initialized.

```
dimension tf(imax,jmax,5)
```

First set everything to zero at $t = 0^-$.

```
do k = 1, 5
  do j = 1, jmx
    do i = 1, imx
      tf(i,j,k) = 0.
    end do
  end do
end do
```

Set the initial pressure to $P_{ref} = 1$.

```
do j = 1, jmx
  do i = 1, imx
    tf(i,j,4) = 1.
  end do
end do
```

Now impulsively start the upper surface with speed $U = u_{big}$, set in `cavity.inp`.

```
do i = 1, imx
  tf(i,jmx,1) = u           ! u-velocity
  tf(i,jmx,3) = 2. * u * dxfac / dx(i,jmax,2) ! vorticity
end do

return
end
```


Timestep

`timestep`, called from `cavity`, determines the Δt timestep size for either local or global timestepping. Δt is then incorporated into the parameters `alam` and `sig`. `alamx` and `sigx` are analogous to convective and diffusive CFL numbers, respectively, and are set in `cavity.inp`.

```
subroutine timestep ( alamx, sigx, re ) !....timestep
```

```
include 'params.inc'
```

Initialize `dtmin = min(Δt)` for global timestep option.

```
dtmin = 2. * alamx * max( dx(2,2,1), dx(2,2,2) )
```

```
do j = 1, jmax
  do i = 1, imax
```

Compute the local convective Δt .

$$\Delta t = 2\lambda_{max} \min(\Delta x, \Delta y)$$

```
dt1 = 2. * alamx * min( dx(i,j,1), dx(i,j,2) )
```

Compute the local diffusive Δt .

$$\Delta t = R_e \sigma_{max} \min(\Delta x^2, \Delta y^2)$$

```
dt2 = re * sigx * min( dx(i,j,1)**2, dx(i,j,2)**2 )
```

The local Δt is the minimum of these timesteps. For global timestepping Δt is the minimum of all the local timesteps.

```
dt = min( dt1, dt2 )
dtmin = min( dtmin, dt )
```

Now incorporate Δt into `alam` and `sig` coefficients.

$$\lambda_x = \frac{\Delta t}{2\Delta x}, \quad \lambda_y = \frac{\Delta t}{2\Delta y}$$

$$\sigma_x = \frac{\Delta t}{R_e \Delta x^2}, \quad \sigma_y = \frac{\Delta t}{R_e \Delta y^2}$$

```
do k = 1, 2
  alam(i,j,k) = dt * .5 / dx(i,j,k)
  sig(i,j,k) = dt / re / dx(i,j,k)**2
end do
```

```
end do
end do
```

This section sets a global timestep for all points. If local timestepping is desired, comment out the following do loops.

```
do j = 1, jmax
  do i = 1, imax
    do k = 1, 2
      alam(i,j,k) = dtmin * .5 / dx(i,j,k)
      sig(i,j,k) = dtmin / re / dx(i,j,k)**2
    end do
  end do
end do

return
end
```

Solvr

Linear system solver for $Af = b$. Called by `cavity` and `multig`. Matrix A is filled in subroutine `loada` and the right-hand side (RHS) vector b is assigned in subroutine `loadb`. Subroutines `sgs`, `bc`, `decp`, and `stopping` are also called, and follow `solvr` in this printout. $Af = b$ is solved iteratively in the time variable using symmetric Gauss-Seidel sweeps.

```
mg = 0  working on top level of multigrid (fine grid)
mg = 1  descending cycle of error smoothing
mg = 2  ascending cycle of error smoothing
```

```
subroutine solvr ( niterp, mg, imx, jmx, igov, dxfac, iskp,
$   nitsgs, re, ubig, resid0, istop, convg, nout, iskpout,
$   cpustart, tf )                                     !....solvr
```

```
include 'params.inc'
logical istop
dimension fold(imax, jmax, 5), tf(imax,jmax,5)
```

Evolve in time until convergence or `niterp` iterations (main loop).

```
do n = 1, niterp
  ntot = ntot + 1
```

Retain previous solution to check convergence rate.

```
do k = 1, 5
  do j = 1, jmx
    do i = 1, imx
      fold(i,j,k) = tf(i,j,k)
    end do
  end do
end do
```

Solve convection/diffusion equations.

$$f_t + uf_x + vf_y - \frac{1}{R_c} \nabla^2 f = -g$$

First step is to load penta-diagonal terms of A matrix in $a_{i,j}$, $a_{i+1,j}$, $a_{i-1,j}$, $a_{i,j+1}$, and $a_{i,j-1}$.

```
call loada ( 1, imx, jmx, dxfac, iskp, mg, tf(1,1,1) )
```

If we are solving the primitive variables formulation.

```
if ( igov .eq. 1 ) then
```

Solve for u from, $u_t + uu_x + vu_y = -P_x + \frac{1}{R_c} \nabla^2 u$.

```
kk = 1
```

Load RHS b vector if evolving on fine grid or the residual if this is a multi-grid error smoothing on a coarse grid.

```

if ( mg .eq. 0 ) then
  call loadb ( kk, imx, jmx, iskp, dxfac, tf(1,1,1) )
else
  if ( mg .eq. 1 ) ll = 1
  if ( mg .eq. 2 ) ll = 2
  do jj = 2, jmx-1
    do ii = 2, imx-1
      b(ii,jj) = res(ii,jj,kk,ll)
    end do
  end do
end if

```

Now perform the symmetric Gauss-Seidel sweeps.

```

call sgs ( kk, nitsgs, imx, jmx, tf(1,1,1) )

```

Save the residual.

```

if ( mg .eq. 0 ) ll = 1
if ( mg .eq. 1 ) ll = 3
do jj = 2, jmx-1
  do ii = 2, imx-1
    res(ii,jj,kk,ll) = b(ii,jj)
$      - aij(ii,jj) * tf(ii,jj,kk)
$      - aip1(ii,jj) * tf(ii+1,jj,kk)
$      - aim1(ii,jj) * tf(ii-1,jj,kk)
$      - ajp1(ii,jj) * tf(ii,jj+1,kk)
$      - ajm1(ii,jj) * tf(ii,jj-1,kk)
  end do
end do

```

Now solve for v from, $v_t + uv_x + vv_y = -P_y + \frac{1}{R_\tau} \nabla^2 v$.

```

kk=2
if ( mg .eq. 0 ) then
  call loadb ( kk, imx, jmx, iskp, dxfac, tf(1,1,1) )
else
  if ( mg .eq. 1 ) ll = 1
  if ( mg .eq. 2 ) ll = 2
  do jj = 2, jmx-1
    do ii = 2, imx-1
      b(ii,jj) = res(ii,jj,kk,ll)
    end do
  end do
end if

```

```

call sgs ( kk, nitsgs, imx, jmx, tf(1,1,1) )
if ( mg .eq. 0 ) ll = 1
if ( mg .eq. 1 ) ll = 3
do jj = 2, jmx-1
  do ii = 2, imx-1
    res(ii,jj,kk,ll) = b(ii,jj)
$      - aij(ii,jj) * tf(ii,jj,kk)
$      - aip1(ii,jj) * tf(ii+1,jj,kk)
$      - aim1(ii,jj) * tf(ii-1,jj,kk)
$      - ajp1(ii,jj) * tf(ii,jj+1,kk)
$      - ajm1(ii,jj) * tf(ii,jj-1,kk)
  end do
end do

```

If instead of primitive variables we are solving the stream-function and vorticity equations, we evolve ζ from. $\zeta_t + u\zeta_x + v\zeta_y = \frac{1}{Rv}\nabla^2\zeta$.

```

else if ( igov .eq. 2 ) then

  kk = 3
  if ( mg .eq. 0 ) then
    call loadb ( kk, imx, jmx, iskp, dxfac, tf(1,1,1) )
  else
    if ( mg .eq. 1 ) ll = 1
    if ( mg .eq. 2 ) ll = 2
    do jj = 2, jmx-1
      do ii = 2, imx-1
        b(ii,jj) = res(ii,jj,kk,ll)
      end do
    end do
  end if
  call sgs ( kk, nitsgs, imx, jmx, tf(1,1,1) )
  if ( mg .eq. 0 ) ll = 1
  if ( mg .eq. 1 ) ll = 3
  do jj = 2, jmx-1
    do ii = 2, imx-1
      res(ii,jj,kk,ll) = b(ii,jj)
$      - aij(ii,jj) * tf(ii,jj,kk)
$      - aip1(ii,jj) * tf(ii+1,jj,kk)
$      - aim1(ii,jj) * tf(ii-1,jj,kk)
$      - ajp1(ii,jj) * tf(ii,jj+1,kk)
$      - ajm1(ii,jj) * tf(ii,jj-1,kk)
    end do
  end do

else
  write (6,*) ' bad value for igov = ', igov

```

```
endif
```

Having evolved the convective/diffusive equations, the Poisson equations remain to be solved.

$$-\nabla^2 f = -g$$

Load penta-diagonal terms for A matrix.

```
call loada ( 2, imx, jmx, dxfac, iskp, mg, tf(1,1,1) )
```

If using the primitive variables formulation we solve for P .

```
if ( igov .eq. 1 ) then

  kk = 4
  if ( mg .eq. 0 ) then
    call loadb ( kk, imx, jmx, iskp, dxfac, tf(1,1,1) )
  else
    if ( mg .eq. 1 ) ll = 1
    if ( mg .eq. 2 ) ll = 2
    do jj = 2, jmx-1
      do ii = 2, imx-1
        b(ii,jj) = res(ii,jj,kk,ll)
      end do
    end do
  end if
  call sgs ( kk, nitsgs, imx, jmx, tf(1,1,1) )
  if ( mg .eq. 0 ) ll = 1
  if ( mg .eq. 1 ) ll = 3
  do jj = 2, jmx-1
    do ii = 2, imx-1
      res(ii,jj,kk,ll) = b(ii,jj)
      $      - aij(ii,jj) * tf(ii,jj,kk)
      $      - aip1(ii,jj) * tf(ii+1,jj,kk)
      $      - aim1(ii,jj) * tf(ii-1,jj,kk)
      $      - ajp1(ii,jj) * tf(ii,jj+1,kk)
      $      - ajm1(ii,jj) * tf(ii,jj-1,kk)
    end do
  end do
```

Otherwise solve for the stream-function ψ from, $\nabla^2 \psi = \zeta$.

```
else
  kk = 5
  if ( mg .eq. 0 ) then
    call loadb ( kk, imx, jmx, iskp, dxfac, tf(1,1,1) )
  else
    if ( mg .eq. 1 ) ll = 1
    if ( mg .eq. 2 ) ll = 2
```

```

do jj = 2, jmx-1
  do ii = 2, imx-1
    b(ii,jj) = res(ii,jj,kk,ll)
  end do
end do
end if
call sgs ( kk, nitsgs, imx, jmx, tf(1,1,1) )
if ( mg .eq. 0 ) ll = 1
if ( mg .eq. 1 ) ll = 3
do jj = 2, jmx-1
  do ii = 2, imx-1
    res(ii,jj,kk,ll) = b(ii,jj)
$      - aij(ii,jj) * tf(ii,jj,kk)
$      - aip1(ii,jj) * tf(ii+1,jj,kk)
$      - aim1(ii,jj) * tf(ii-1,jj,kk)
$      - ajp1(ii,jj) * tf(ii,jj+1,kk)
$      - ajm1(ii,jj) * tf(ii,jj-1,kk)
  end do
end do
end if

```

If we are performing an update on the fine mesh then update the boundary points. Also, for the stream-function/vorticity formulation the velocities and pressure need to be decoded at the interior points. Otherwise, if this is an error smoothing iteration, skip this section.

```

if ( mg .eq. 0 ) then

  call bc ( igov, imx, jmx, ubig, re, iskp, dxfac,
$          tf(1,1,1) )

  if ( igov .eq. 2 ) then

```

Decode the velocities.

```

do j = 2, jmx-1
  j1 = ( j - 1 ) * iskp + 1
  do i = 2, imx-1
    i1 = ( i - 1 ) * iskp + 1
    tf(i,j,1) = .5 * ( tf(i,j+1,5) - tf(i,j-1,5) )
$      / dx(i1,j1,2) * dxfac
    tf(i,j,2) = .5 * ( tf(i-1,j,5) - tf(i+1,j,5) )
$      / dx(i1,j1,1) * dxfac
  end do
end do

```

Decode the pressure by iterating on the Poisson equation.

```

        call decp ( imx, jmx, iskp, tf(1,1,1) )
    end if
end if

```

Check convergence criteria to see if we stop.

```

    call stopping ( n, imx, jmx, igov, resid0, istop, convg,
$      resid, nout, iskpout, cpustart, iskp, mg,
$      fold(1,1,1), tf(1,1,1) )

```

Jump out of loop when converged.

```

    if ( istop .eq. .true. ) go to 100

```

End of the main loop for time evolution.

```

    end do

    idebugwrite = 0
    if ( idebugwrite .eq. 1 ) write (6,200)
200 format (/ 'reached maximum number of iterations without ',
$      'converging'/)

100 continue
    if ( idebugwrite .eq. 1 ) write (6,201) n, iskp, resid
201 format ( ' n = ', i5, ' iskp = ', i2, ' resid = ', e13.5 )

    return
end

```


Load A

Form the penta-diagonal Left-Hand Side (LHS) implicit matrix. Stored in five diagonal vectors, $A_{i,j}$, $A_{i\pm 1,j}$, and $A_{i,j\pm 1}$. `loada` is called from `solvr`.

```

subroutine loada ( ieq, imx, jmx, dxfac, iskp, mg, tf )

include 'params.inc'
dimension tf(imax,jmax,5)

d1 = dxfac
d2 = dxfac**2

```

For the convective/diffusive equations for u , v , or ζ ,

$$A_{i,j} = 1 + 2\sigma_x + 2\sigma_y$$

$$A_{i\pm 1,j} = -\sigma_x \pm \lambda_x u, \quad A_{i,j\pm 1} = -\sigma_y \pm \lambda_y v$$

```

if ( ieq .eq. 1 ) then
do j = 1, jmx
j1 = ( j - 1 ) * iskp + 1
do i = 1, imx
i1 = ( i - 1 ) * iskp + 1
if ( mg .eq. 0 ) then
u1 = tf(i,j,1)
v1 = tf(i,j,2)
else
u1 = f(i1,j1,1)
v1 = f(i1,j1,2)
end if
aij(i,j) = 1. + 2.*sig(i1,j1,1)*d2 + 2.*sig(i1,j1,2)*d2
aip1(i,j) = -sig(i1,j1,1)*d2 + alam(i1,j1,1)*d1 * u1
aim1(i,j) = -sig(i1,j1,1)*d2 - alam(i1,j1,1)*d1 * u1
ajp1(i,j) = -sig(i1,j1,2)*d2 + alam(i1,j1,2)*d1 * v1
ajm1(i,j) = -sig(i1,j1,2)*d2 - alam(i1,j1,2)*d1 * v1
end do
end do

```

For the Poisson equations for P or ψ ,

$$A_{i,j} = \frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}$$

$$A_{i\pm 1,j} = -\frac{1}{\Delta x^2}, \quad A_{i,j\pm 1} = -\frac{1}{\Delta y^2}$$

```

else if ( ieq .eq. 2 ) then
do j = 1, jmx

```

```

      j1 = ( j - 1 ) * iskp + 1
      do i = 1, imx
        i1 = ( i - 1 ) * iskp + 1
        aij (i,j) = 2. * d2 / dx(i1,j1,1)**2 +
$          2.* d2 / dx(i1,j1,2)**2
        aip1(i,j) = -d2 / dx(i1,j1,1)**2
        aim1(i,j) = -d2 / dx(i1,j1,1)**2
        ajp1(i,j) = -d2 / dx(i1,j1,2)**2
        ajm1(i,j) = -d2 / dx(i1,j1,2)**2
      end do
    end do

  end if

  return
end

```

Load b

Assign the Right-Hand Side (RHS) vector b for advancing $Af = b$ from time level n to level $n + 1$. loadb is called by solvr.

```
subroutine loadb ( mf, imx, jmx, iskp, dxfac, tf )

include 'params.inc'

dimension tf(imax,jmax,5)

d1 = dxfac
d2 = dxfac * dxfac
```

If solving for u ,

$$b = u - \left(\frac{\Delta t}{\Delta x} \right) \frac{P_{i+1} - P_{i-1}}{2}$$

```
if ( mf .eq. 1 ) then
do j = 2, jmx - 1
j1 = ( j - 1 ) * iskp + 1
do i = 2, imx - 1
i1 = ( i - 1 ) * iskp + 1
b(i,j) = tf(i,j,mf) - alam(i1,j1,mf) * dxfac *
$ ( tf(i+1,j,4) - tf(i-1,j,4) )
end do
end do
```

If solving for v ,

$$b = v - \left(\frac{\Delta t}{\Delta y} \right) \frac{P_{j+1} - P_{j-1}}{2}$$

```
else if ( mf .eq. 2 ) then
do j = 2, jmx - 1
j1 = ( j - 1 ) * iskp + 1
do i = 2, imx - 1
i1 = ( i - 1 ) * iskp + 1
b(i,j) = tf(i,j,mf) - alam(i1,j1,mf) * dxfac *
$ ( tf(i,j+1,4) - tf(i,j-1,4) )
end do
end do
```

If solving for ζ , $b = \zeta$.

```
else if ( mf .eq. 3 ) then
do j = 2, jmx - 1
do i = 2, imx - 1
b(i,j) = tf(i,j,mf)
end do
end do
```

If solving for P ,

$$b = -\frac{C}{\Delta t} + \frac{(u_{i+1} - u_{i-1})^2}{4\Delta x^2} + \frac{(u_{j+1} - u_{j-1})(v_{i+1} - v_{i-1})}{2\Delta x \Delta y} + \frac{(v_{j+1} - v_{j-1})^2}{4\Delta y^2} - \frac{1}{R_e} \left[\frac{C_{i+1} - 2C + C_{i-1}}{\Delta x^2} + \frac{C_{j+1} - 2C + C_{j-1}}{\Delta y^2} \right]$$

where,

$$C = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + \frac{v_{j+1} - v_{j-1}}{2\Delta y}$$

```

else if ( mf .eq. 4 ) then
  do j = 2, jmx - 1
    j1 = ( j - 1 ) * iskp + 1
    do i = 2, imx - 1
      i1 = ( i - 1 ) * iskp + 1
      b(i,j) = .25 * d2*( (tf(i+1,j,1) -
$         tf(i-1,j,1))**2 / dx(i1,j1,1)**2
$         + 2.*(tf(i,j+1,1) - tf(i,j-1,1)) * (tf(i+1,j,2)
$         - tf(i-1,j,2)) / dx(i1,j1,1) / dx(i1,j1,2)
$         + (tf(i,j+1,2) - tf(i,j-1,2))**2 /
$         dx(i1,j1,2)**2 )
c      b(i,j) = .5 * (
c      $         ( f(i,j+1,1) - f(i,j-1,1) ) *
c      $         ( f(i+1,j,2) - f(i-1,j,2) ) -
c      $         ( f(i+1,j,1) - f(i-1,j,1) ) *
c      $         ( f(i,j+1,2) - f(i,j-1,2) ) )
c      $         / dx(i,j,1) / dx(i,j,2)
c      b(i,j) = .5 * ( f(i+1,j,1) - f(i-1,j,1) )**2 /
c      $         dx(i,j,1)**2 + .5 * ( f(i,j+1,1) - f(i,j-1,1) ) *
c      $         ( f(i+1,j,2) - f(i-1,j,2) ) / dx(i,j,1) / dx(i,j,2)
      end do
    end do

```

If solving for ψ , $b = -\zeta$.

```

else if ( mf .eq. 5 ) then
  do j = 2, jmx - 1
    do i = 2, imx - 1
      b(i,j) = - tf(i,j,3)
    end do
  end do

else
  write (6,*) ' mf not .le. 5 in loadb, mf = ', mf
end if
c

```

```
c      * diagnostic *
c      do i = 2, imx-1
c      do j = 2, jmx-1
c      write (6,*) ' b=',b(i,j),' for i=',i,' j=',j
c      end do
c      end do

      return
      end
```

Sgs

Symmetric Gauß-Seidel iteration module. Iterates on $Af = b$, sweeping first on increasing i nested inside increasing j , then sweeps again on decreasing j nested inside decreasing i . `niter` symmetric sweeps are performed. The variable/equation to be updated is designated by `mf`. `sgs` is called by `solvr` and `decp`.

```
subroutine sgs ( mf, niter, imx, jmx, tf )
```

```
include 'params.inc'  
dimension tf(imax,jmax,5)
```

```
imx1 = imx - 1  
jmx1 = jmx - 1
```

How many symmetric sweeps to perform--`niter`.

```
do k = 1, niter
```

Forward sweep i and j increasing. Since A is penta-diagonal, the update formula is,

$$f_{i,j} = \frac{1}{a_{i,j}} (b_{i,j} - a_{i+1,j}f_{i+1,j} - a_{i-1,j}f_{i-1,j} - a_{i,j+1}f_{i,j+1} - a_{i,j-1}f_{i,j-1})$$

```
do j = 2, jmx1  
  jpl = j + 1  
  jml = j - 1  
  do i = 2, imx1  
    ip1 = i + 1  
    im1 = i - 1  
    tf(i,j,mf) = ( b(i,j) - aip1(i,j) * tf(ip1,j,mf) -  
$           aim1(i,j) * tf(im1,j,mf) - ajpl(i,j) * tf(i,jpl,mf)  
$           - ajml(i,j) * tf(i,jml,mf) ) / aij(i,j)  
  end do  
end do
```

Backward sweep-- i and j decreasing.

```
do i = imx1, 2, -1  
  ip1 = i + 1  
  im1 = i - 1  
  do j = jmx1, 2, -1  
    jpl = j + 1  
    jml = j - 1  
    tf(i,j,mf) = ( b(i,j) - aip1(i,j) * tf(ip1,j,mf) -  
$           aim1(i,j) * tf(im1,j,mf) - ajpl(i,j) * tf(i,jpl,mf)  
$           - ajml(i,j) * tf(i,jml,mf) ) / aij(i,j)
```

```
    end do  
  end do  
  
end do  
  
return  
end
```

Boundary Conditions

Explicitly update the boundary points. All boundaries are solid, no-slip walls. Upper boundary translates at speed ubig. bc is called by solvr.

```

subroutine bc ( ieq, imx, jmx, ubig, re, iskp, dxfac, tf )

include 'params.inc'
dimension tf(imax,jmax,5)

d2 = dxfac * dxfac

i = 1, left wall

i = 1
i1 = ( i - 1 ) * iskp + 1
do j = 2, jmx-1
  j1 = ( j - 1 ) * iskp + 1
c      if ( ieq .eq. 1 ) then      ! primitive variables formulation
u = v = 0

      tf(i,j,1) = 0.
      tf(i,j,2) = 0.

```

Several options for pressure are:

$$P_{wall} = P_{wall+1}$$

$$P_{wall} = \frac{4}{3}P_{wall+1} - \frac{1}{3}P_{wall+2}$$

$$P_{wall} = \frac{1}{3} \left(4P_{wall+1} - P_{wall+2} + \frac{4P_{wall+1} - 2P_{wall+2}}{Re \Delta x} + 2P_{wall+1}^2 \right)$$

$$P_{wall} = P_{wall+1} + \frac{2P_{wall+1} - P_{wall+2}}{Re \Delta x} + P_{wall+1}^2$$

```

      tf(i,j,4) = tf(i+1,j,4)
c      f(i,j,4) = 4. / 3. * f(i+1,j,4) - f(i+2,j,4) / 3.
c      f(i,j,4) = ( 4. * f(i+1,j,4) - f(i+2,j,4) +
c      $          ( 4. * f(i+1,j,1) - 2. * f(i+2,j,1) )
c      $          / re / dx(i,j,1) + 2. * f(i+1,j,1)**2 ) / 3.
c      f(i,j,4) = f(i+1,j,4) + ( 2. * f(i+1,j,1) - f(i+2,j,1) )
c      $          / re / dx(i,j,1) + f(i+1,j,1)**2
c
c      else      ! vorticity/stream function formulation

```

Options for ψ and ζ are:

$$\psi_{wall} = 0$$

$$\psi_{wall} = \psi_{wall+1}$$

$$\psi_{wall} = \frac{4}{3}\psi_{wall+1} - \frac{1}{3}\psi_{wall+2}$$

$$\zeta_{wall} = \frac{2\psi_{wall+1}}{\Delta x^2}$$

$$\zeta_{wall} = \frac{1}{\Delta x^2} (\psi_{wall} - 2\psi_{wall+1} + \psi_{wall+2})$$

```

tf(i,j,5) = 0.
c f(i,j,5) = 4. / 3. * f(i+1,j,5) - f(i+2,j,5) / 3.
c f(i,j,5) = f(i+1,j,5)

tf(i,j,3) = 2. * d2 * tf(i+1,j,5) / dx(i1,j1,1)**2
c f(i,j,3) = ( f(i,j,5) - 2. * f(i+1,j,5) + f(i+2,j,5) ) /
c $ dx(i,j,1)**2
c end if
end do

```

i = imx, right wall

```

i = imx
i1 = ( i - 1 ) * iskp + 1
do j = 2, jmx-1
  j1 = ( j - 1 ) * iskp + 1
c if ( ieq .eq. 1 ) then
  tf(i,j,1) = 0.
  tf(i,j,2) = 0.
c f(i,j,4) = 4. / 3. * f(i-1,j,4) - f(i-2,j,4) / 3.
c f(i,j,4) = f(i-1,j,4) - ( 2. * f(i-1,j,1) - f(i-2,j,1) )
c $ / re / dx(i,j,1) + f(i-1,j,1)**2
c tf(i,j,4) = tf(i-1,j,4)
c else
c f(i,j,5) = 4. / 3. * f(i-1,j,5) - f(i-2,j,5) / 3.
c f(i,j,5) = f(i-1,j,5)
c tf(i,j,5) = 0.
c f(i,j,3) = ( f(i,j,5) - 2. * f(i-1,j,5) + f(i-2,j,5) ) /
c $ dx(i,j,1)**2
c tf(i,j,3) = 2. * d2 * tf(i-1,j,5) / dx(i1,j1,1)**2
c end if
end do

```

j = 1, bottom

```

j = 1
j1 = ( j - 1 ) * iskp + 1
do i = 2, imx-1
  i1 = ( i - 1 ) * iskp + 1

```

```

c      if ( ieq .eq. 1 ) then
c          tf(i,j,1) = 0.
c          tf(i,j,2) = 0.
c          f(i,j,4) = 4. / 3. * f(i,j+1,4) - f(i,j+2,4) / 3.
c          f(i,j,4) = f(i,j+1,4) + ( 2. * f(i,j+1,2) - f(i,j+2,2) )
c          $          / re / dx(i,j,2) + f(i,j+1,2)**2
c          tf(i,j,4) = tf(i,j+1,4)
c      else
c          f(i,j,5) = 4. / 3. * f(i,j+1,5) - f(i,j+2,5) / 3.
c          f(i,j,5) = f(i,j+1,5)
c          tf(i,j,5) = 0.
c          f(i,j,3) = ( f(i,j,5) - 2. * f(i,j+1,5) + f(i,j+2,5) ) /
c          $          dx(i,j,2)**2
c          tf(i,j,3) = 2. * d2 * tf(i,j+1,5) / dx(i1,j1,2)**2
c      end if
c  end do

```

j = jmx, top

```

j = jmx
j1 = ( j - 1 ) * iskp + 1
do i = 2, imx-1
    i1 = ( i - 1 ) * iskp + 1
c      if ( ieq .eq. 1 ) then

```

Same options for v and P . $u = U$ (ubig).

$$\psi_{wall} = 0 \text{ or, } \psi_{wall} = \frac{1}{3} (4\psi_{wall-1} - \psi_{wall-2} + 2\Delta y U)$$

$$\zeta_{wall} = \frac{2}{\Delta y^2} (\psi_{wall-1} + \Delta y U) \text{ or, } \zeta_{wall} = \frac{1}{\Delta y^2} (\psi_{wall} - 2\psi_{wall-1} + \psi_{wall-2})$$

```

c          tf(i,j,1) = ubig
c          tf(i,j,2) = 0.
c          f(i,j,4) = 4. / 3. * f(i,j-1,4) - f(i,j-2,4) / 3.
c          f(i,j,4) = f(i,j-1,4) - ( 2. * f(i,j-1,2) - f(i,j-2,2) )
c          $          / re / dx(i,j,2) + f(i,j-1,2)**2
c          tf(i,j,4) = tf(i,j-1,4)
c      else
c          f(i,j,5) = ( 4. * f(i,j-1,5) - f(i,j-2,5) +
c          $          2. * ubig * dx(i,j,2) ) / 3.
c          tf(i,j,5) = 0.
c          f(i,j,3) = ( f(i,j,5) - 2. * f(i,j-1,5) + f(i,j-2,5) ) /
c          $          dx(i,j,2)**2
c          tf(i,j,3) = 2. * d2 * ( tf(i,j-1,5) +
$          ubig * dx(i1,j1,2) / dxfac ) / dx(i1,j1,2)**2
c      end if
c  end do

```

just set the corner points to a neighboring point

```
do k = 1, 5
  tf(1,1,k) = tf(2,1,k)
  tf(imx,1,k) = tf(imx-1,1,k)
  tf(1,jmx,k) = tf(2,jmx,k)
  tf(imx,jmx,k) = tf(imx-1,jmx,k)
end do

return
end
```

Decp

Decode the pressure from the stream-function and vorticity. Relies on the fact that the velocities have already been decoded in `solvr`. The Poisson equation for the steady-state pressure,

$$\nabla^2 P = 2(u_x v_y - u_y v_x)$$

is relaxed using symmetric Gauß-Seidel sweeps.

Called from `cavity`. Calls `sgs`.

```
subroutine decp ( imx, jmx, iskp, tf )
include 'params.inc'
dimension tf( imax, jmax, 5 )
```

Build pentadiagonal LHS and forcing function RHS at interior points. Multiply through by grid spacing for convenience, giving,

$$\Delta x^2 \Delta y^2 \nabla^2 P = 2 \Delta x^2 \Delta y^2 (u_x v_y - u_y v_x)$$

```
do i = 2, imx - 1
  i1 = ( i - 1 ) * iskp + 1
  do j = 2, jmx - 1
    j1 = ( j - 1 ) * iskp + 1
```

Construct LHS with two-dimensional, second-order central differences,

$$\Delta x^2 \Delta y^2 \nabla^2 P_{i,j} \rightarrow \Delta y^2 (P_{i+1} + P_{i-1}) + \Delta x^2 (P_{j+1} + P_{j-1}) - 2(\Delta x^2 + \Delta y^2) P_{i,j}$$

```
  aij(i,j) = -2. * ( dx(i1,j1,1)**2 + dx(i1,j1,2)**2 )
  aip1(i,j) = dx(i1,j1,2)**2
  aim1(i,j) = dx(i1,j1,2)**2
  ajp1(i,j) = dx(i1,j1,1)**2
  ajm1(i,j) = dx(i1,j1,1)**2
```

Construct the RHS vector using second-order central differences,

$$2 \Delta x^2 \Delta y^2 (u_x v_y - u_y v_x) \rightarrow$$

$$\frac{\Delta x \Delta y}{2} [(u_{i+1} - u_{i-1})(v_{j+1} - v_{j-1}) - (u_{j+1} - u_{j-1})(v_{i+1} - v_{i-1})]$$

```
  ux = f(i+1,j,1) - f(i-1,j,1)
  uy = f(i,j+1,1) - f(i,j-1,1)
  vx = f(i+1,j,2) - f(i-1,j,2)
  vy = f(i,j+1,2) - f(i,j-1,2)
  b(i,j) = .5 * dx(i1,j1,1) * dx(i1,j1,2) *
$ ( ux * vy - uy * vx )
  end do
end do
```

Finally iteratively solve for the pressure using Gauß-Seidel sweeps.

```
call sgs ( 4, 1, imx, jmx, tf(1,1,1) )  
return  
end
```

Stopping

Check to see if convergence criteria has been met. Also outputs convergence and timing history. Called by solvr.

```
subroutine stopping ( n, imx, jmx, igov, resid0, istop,  
$   convg, resid, nout, iskpout, cpustart, iskp, mg,  
$   fold, tf )
```

```
include 'params.inc'  
dimension fold(imax,jmax,5), tf(imax,jmax,5)  
logical istop
```

Compute L_2 residual of all variables at all points.

```
resid = 0.  
do j = 1, jmx  
  do i = 1, imx  
c    primitive variables  
    if ( igov .eq. 1 ) then  
      resid = resid + ( tf(i,j,1) - fold(i,j,1) )**2  
      resid = resid + ( tf(i,j,2) - fold(i,j,2) )**2  
      resid = resid + ( tf(i,j,4) - fold(i,j,4) )**2  
    else  
c    stream-function/vorticity  
      resid = resid + ( tf(i,j,3) - fold(i,j,3) )**2  
      resid = resid + ( tf(i,j,5) - fold(i,j,5) )**2  
    end if  
  end do  
end do  
resid = sqrt( resid )
```

Initialize starting residual if first iteration.

```
if ( resid0 .lt. 0. ) then  
  resid0 = resid  
  write (45) 1, etime(a1,a2)-cpustart, 1.  
  write (6,*) ' resid0 = ', resid0  
  nout = 1  
end if
```

Normalize the residual by the starting residual.

```
resid = resid / resid0
```

Check for convergence if on fine mesh.

```
if ( iskp .eq. 1 .and. abs(resid) .lt. convg ) then  
  istop = .true.  
  write (6,*) ' *** Solution Converged ***'  
end if
```

Save convergence history if on fine mesh.

```
    if ( ( mod( ntot, iskpout ) .eq. 0
$      .or. istop .eq. .true. ) .and. mg .eq. 0 )then
        cpu1 = etime( atime1, atime2 ) - cpustart
        write(45) ntot, cpu1, resid
        write (6,110) ntot, iskp, cpu1, resid
110    format ( 'iter = ', i5, ', iskp = ', i2,', cput = ',
$          f7.1, 's,', ' resid = ', e12.6 )
        nout = nout + 1
    end if

    return
end
```

Restrict

Restrict a fine-mesh vector to a coarse mesh by simple injection of the overlapping points. `vbig` is the vector to be restricted, defined on the fine grid. `vsmal` is the resultant vector, defined on the coarse grid. `ix` and `jx` are dimensions of `vsmal`. Called from `cavity` and `multig`.

```
subroutine restrict ( ix, jx, vbig, vsmal )

include 'params.inc'
dimension vbig(imax,jmax,5), vsmal(imax,jmax,5)

do k = 1, 5
  do j = 1, jx
    j2 = 2 * j - 1
    do i = 1, ix
      i2 = 2 * i - 1
      vsmal(i,j,k) = vbig(i2,j2,k)
    end do
  end do
end do

return
end
```


Prolong

A linear-averaging prolongation of a vector from a coarse grid to a fine grid. `vsmal` is the vector to prolongate, defined on the coarse grid. `vbig` is the result of the prolongation, defined on the fine grid. `ix` and `jx` are dimensions of `vsmal`. Called by `cavity` and `multig`.

```
subroutine prolong ( ix, jx, vbig, vsmal )

include 'params.inc'
dimension vbig(imax,jmax,5), vsmal(imax,jmax,5)

do k = 1, 5
```

Simple injection for the overlapping points.

```
do j = 1, jx
  j2 = 2 * j - 1
  do i = 1, ix
    i2 = 2 * i - 1
    vbig(i2,j2,k) = vsmal(i,j,k)
  end do
end do
```

Two-point averaging in i for the new fine-grid points along a $j = \text{const}$ line of the coarse grid.

```
do j = 1, jx
  j2 = 2 * j - 1
  do i = 1, ix-1
    i2 = 2 * i
    vbig(i2,j2,k) = .5 * ( vsmal(i,j,k) + vsmal(i+1,j,k) )
  end do
end do
```

Two-point averaging in j for the new fine-grid points along a $i = \text{const}$ line of the coarse grid.

```
do i = 1, ix
  i2 = 2 * i - 1
  do j = 1, jx-1
    j2 = 2 * j
    vbig(i2,j2,k) = .5 * ( vsmal(i,j,k) + vsmal(i,j+1,k) )
  end do
end do
```

Four-point averaging for remaining fine-grid points, corresponding to $i \pm \frac{1}{2}$, $j \pm \frac{1}{2}$ on the coarse grid.

```
do j = 1, jx - 1
  j2 = 2 * j
  do i = 1, ix - 1
    i2 = 2 * i
    vbig(i2,j2,k) = .25 * ( vsmal(i,j,k) + vsmal(i+1,j,k) +
$      vsmal(i,j+1,k) + vsmal(i+1,j+1,k) )
  end do
end do

end do

return
end
```

Multig

V-cycle multigrid driver. Called from `cavity`. Calls `solvr`, `restrict`, and `prolong`.

```
subroutine multig

include 'params.inc'
logical istop

common / parms / alength, alamax, sigmax, re, height, ubig,
$   igov, nitsgs, niterp, convg, iskpout, img, istop,
$   igrd, beta
common / parm2 / cpustart, dxfin, dxmed, dxcor, iskpfin,
$   iskpmed, iskpcor, resid0, nout
```

Begin main loop of V-cycle. Perform `niterp` cycles.

```
do nn = 1, niterp
```

Relax solution on fine-mesh. First argument is the number of smoothing iterations to perform.

```
call solvr ( 3, 0, imax, jmax, igov, dxfin, iskpfin,
$   nitsgs, re, ubig, resid0, istop, convg, nout,
$   iskpout, cpustart, f(1,1,1) )
```

Are we converged on the fine mesh?

```
if ( istop .eq. .true. ) go to 200
```

Restrict residual from fine mesh to medium mesh.

```
call restrict(imxmed,jmxmed,res(1,1,1,1),res(1,1,1,2))
do k = 1, 5
do j = 1, jmxmed
do i = 1, imxmed
res(i,j,k,1) = res(i,j,k,2)
fmed(i,j,k) = 0.    ! zero out error on medium mesh
end do
end do
end do
```

Smooth error on medium mesh. Again, the first argument is the number of smoothing iterations to perform.

```
call solvr ( 2, 1, imxmed, jmxmed, igov, dxmed, iskpmed,
$   nitsgs, re, ubig, resid0, istop, convg, nout,
$   iskpout, cpustart, fmed(1,1,1) )
```

Restrict residual from medium to coarse mesh.

```

call restrict(imxcor,jmxcor,res(1,1,1,3),res(1,1,1,1))
do k = 1, 5
  do j = 1, jmxcor
    do i = 1, imxcor
      fcor(i,j,k) = 0.
    end do
  end do
end do

```

Smooth error on coarse mesh.

```

call solvr ( 1, 1, imxcor, jmxcor, igov, dxcor, iskpcor,
$          nitsgs, re, ubig, resid0, istop, convg, nout,
$          iskpout, cpustart, fcor(1,1,1) )

```

Prolong coarse-mesh error to medium mesh.

```

call prolong (imxcor, jmxcor, res(1,1,1,3), fcor(1,1,1))

```

Correct medium-mesh error with contribution from coarse mesh.

```

do k = 1, 5
  do j = 2, jmxmed-1
    do i = 2, imxmed-1
      fmed(i,j,k) = fmed(i,j,k) + res(i,j,k,3)
    end do
  end do
end do

```

Smooth medium-mesh error again.

```

call solvr ( 2, 2, imxmed, jmxmed, igov, dxmed, iskpmed,
$          nitsgs, re, ubig, resid0, istop, convg, nout,
$          iskpout, cpustart, fmed(1,1,1) )

```

Prolong medium-mesh error to fine mesh.

```

call prolong (imxmed, jmxmed, res(1,1,1,1), fmed(1,1,1))

```

Update fine-mesh solution with correction to residual from medium mesh.

```

do k = 1, 5
  do j = 2, jmax-1
    do i = 2, imax-1
      f(i,j,k) = f(i,j,k) + res(i,j,k,1)
    end do
  end do
end do

```

Smooth fine-mesh solution again.

```
        call solvr ( 3, 0, imax, jmax, igov, dxfin, iskpfin,  
$          nitsgs, re, ubig, resid0, istop, convg, nout,  
$          iskpout, cpustart, f(1,1,1) )
```

Now loop back to repeat V-cycle.

```
        end do  
200    continue
```

```
        return  
        end
```

Outsol

Solution-output routine, called from `cavity`. Solution is written to `cavity_tec.dat` in `TECPLOT[2]` format for variables: $x, y, \Delta x, \Delta y, u, v, \zeta, P, \psi, u_x + v_y$.

Also outputs plot file of convergence history to `cavity_convlg.dat`.

```
subroutine outsol ( imx, jmx, nout )
```

```
include 'params.inc'
```

```
dimension deriv(imax,jmax,2)
```

Compute derivatives of velocities to verify continuity equation. Central difference interior points, one-sided differences on boundaries. Should get $u_x + v_y = 0$.

u_x

```
do j = 1, jmx
  deriv(1,j,1) = ( f(2,j,1) - f(1,j,1) ) / dx(1,j,1)
  do i = 2, imx-1
    deriv(i,j,1) = .5 * ( f(i+1,j,1) - f(i-1,j,1) )
$      / dx(i,j,1)
  end do
  deriv(imx,j,1) = ( f(imx,j,1) - f(imx-1,j,1) )
$      / dx(imx,j,1)
end do
```

v_y

```
do i = 1, imx
  deriv(i,1,2) = ( f(i,2,2) - f(i,1,2) ) / dx(i,1,2)
  do j = 2, jmx-1
    deriv(i,j,2) = .5 * ( f(i,j+1,2) - f(i,j-1,2) )
$      / dx(i,j,2)
  end do
  deriv(i,jmx,2) = ( f(i,jmx,2) - f(i,jmx-1,2) )
$      / dx(i,jmx,2)
end do
```

Tecplot-format grid and solution file.

```
open ( 15, file = 'cavity_tec.dat', form = 'formatted' )
write ( 15, 100 ) imx, jmx
100 format ( 'TITLE = "flowsol" ' /
$   'VARIABLES = "X", "Y", "dx", "dy" ' /
$   '"u", "v", "zeta", ' /
$   '"P", "psi", "cont" ' /
$   'ZONE I=', i3, ', J=', j3, ', F=BLOCK' )

write ( 15, * ) ((( x(i,j,k), i=1,imx), j=1,jmx), k=1,2)
write ( 15, * ) ((( dx(i,j,k), i=1,imx), j=1,jmx), k=1,2)
```

```

write ( 15, * ) ((( f(i,j,k), i=1,imx), j=1,jmx), k=1,5)
write ( 15, * ) ((( deriv(i,j,1) + deriv(i,j,2) ),
$      i=1,imx), j=1,jmx)

```

Tecplot-format convergence history.

```

open ( 16, file = 'cavity_conv.dat', form = 'formatted' )
write ( 16, 101 ) nout
101 format ( 'TITLE = "Convergence"' /
$      'VARIABLES = "iter", "CPU", "R-L_2"' /
$      'ZONE I=', i5, ', ', J=1, F=point' )

rewind (45)
do n = 1, nout
  read (45) i, cpu, resid
  write (16,*) i, cpu, resid
end do

return
end

```

Results

Incompressible Navier-Stokes solutions were computed inside a unit cavity, with the upper surface translating to the right. Three Reynolds numbers (R_e) were considered, 100, 400, and 2000. In all cases uniform meshes were employed, with the same spacing in both x and y directions.

Convergence times for six cases, corresponding to a coarse and fine mesh convergence study at each of the three Reynolds numbers, are tabulated in Table 1. Solution times are seen to increase with increasing R_e , and to increase by an order of magnitude with a quadrupling of the grid density. Solution times reported are for an SGI 195 MHz R10000 CPU. Each case was started from a stationary initial condition, using three-level mesh sequencing and multigrid smoothing to accelerate convergence.

R_e	J	CPU sec.	L_2	Resolution
100	51	13	10^{-6}	± 0.01
100	101	121	10^{-6}	± 0.005
400	101	168	10^{-6}	± 0.005
400	126	224	10^{-6}	± 0.004
2000	125	707	2×10^{-5}	± 0.004
2000	249	4039	5×10^{-5}	± 0.002

Table 1: Cases on unit square with $J \times J$ mesh.

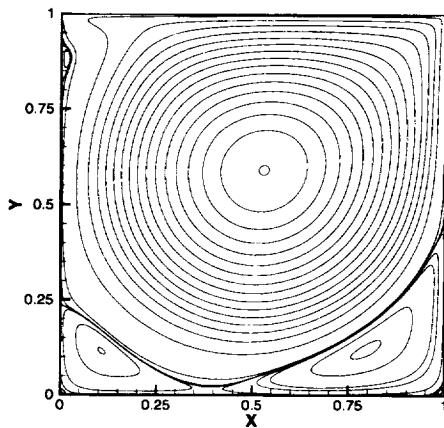


Figure 1: Stream-function contours at $R_e = 2000$, 201×201 mesh.

	Ghia	Schreiber	Present (coarse)	Present (fine)
x_0	0.617	0.617	0.62	0.615
y_0	0.734	0.742	0.74	0.740
ψ_0	-0.103	-0.103	-0.102	-0.103
ζ_0	3.17	3.18	3.18	3.20
x_1	0.031	0.033	0.03	0.035
y_1	0.039	0.025	0.03	0.035
$\psi_1 (\times 10^6)$	1.74	2.05	3.22	2.09
ζ_1	-0.0255	-0.0080	-0.0231	-0.0153
L_1	0.078	--	0.09	0.085
H_1	0.078	--	0.09	0.085
x_2	0.945	0.942	0.94	0.940
y_2	0.063	0.050	0.06	0.060
$\psi_2 (\times 10^5)$	1.25	1.32	1.59	1.33
ζ_2	-0.0331	-0.0255	-0.0356	-0.0359
L_2	0.133	--	0.13	0.135
H_2	0.148	--	0.15	0.155

Table 2: Vortex locations, stream-function, and vorticity for $R_v = 100$.

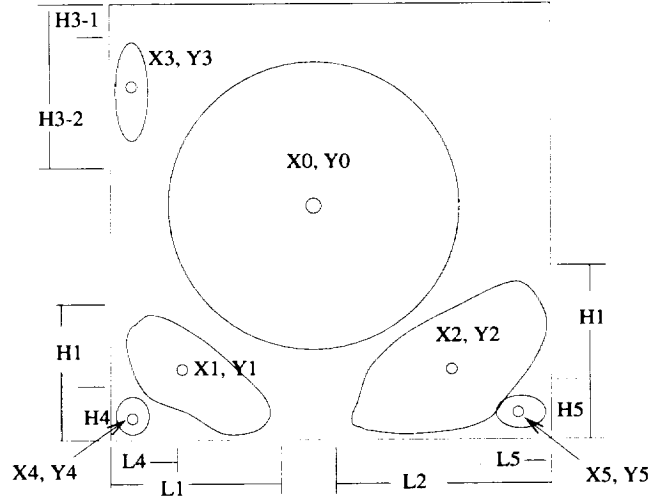


Figure 2: Vortex center and separation point naming conventions.

At $R_e = 2000$, six vortices are predicted, seen in Figure 1. In addition to the primary vortex, there are three secondary vortices in the lower corners and upper-left side. The two remaining tertiary vortices are located in the extreme lower corners. Figure 2 is a cartoon sketch identifying the nomenclature for describing the vortices and their strengths.

Numerical data from the six cases in Table 1 in the form of vortex locations, stream-function and vorticity strengths, and separation points are compared with the results of Ghia *et al*[3] and Schreiber and Keller[4] in Tables 2 and 3. Excellent agreement is seen for $R_e = 100$, Table 2, between the present method and the other sources. At this Reynolds number only three vortices are present.

Table 3 contains the data for $R_e = 400$. Schreiber reports data for only three vortices at this Reynolds number, though the present method resolved four and Ghia reports five vortices. Again, excellent agreement is seen between the data sets. Table 4 lists the results of the grid-convergence study at $R_e = 2000$, where all six vortices appear.

Summary

An annotated source code for the solution of incompressible viscous flow in a two-dimensional cavity is presented. The Navier-Stokes equations are cast in a stream-function/vorticity formulation with second-order spatial accuracy. Convergence to a steady state is accelerated *via* multigrid relaxation.

Solutions are obtained for Reynolds numbers of 100, 400, and 2000. Solution accuracy is checked *versus* previously published numerical results. Excellent agreement is seen in the location of vortex centers, separation points, vorticity strengths, and stream-function values amongst the data sets.

The present code is seen to be an accurate and efficient means for solving the incompressible viscous driven cavity problem.

References

- [1] Wood, W. A., "Multigrid Approach to Incompressible Viscous Cavity Flows," NASA TM 110262, May 1996.
- [2] Amtec Engineering, Inc., Bellevue, Washington, *Tecplot User's Manual: Version 7*, Aug. 1996.
- [3] Ghia, U., Ghia, K. N., and Shin, C. T., "High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method," *Journal of Computational Physics*, Vol. 48, Oct. 1982, pp. 387-411.
- [4] Schreiber, R. and Keller, H. B., "Driven Cavity Flows by Efficient Numerical Techniques," *Journal of Computational Physics*, Vol. 49, 1983, pp. 310-333.

	Ghia	Schreiber	Present (coarse)	Present (fine)
x_0	0.555	0.557	0.555	0.556
y_0	0.606	0.607	0.610	0.608
ψ_0	-0.114	-0.113	-0.112	-0.113
ζ_0	2.29	2.28	2.27	2.29
x_1	0.051	0.050	0.050	0.052
y_1	0.047	0.043	0.045	0.048
$\psi_1 (\times 10^5)$	1.42	1.45	1.47	1.49
ζ_1	-0.0570	-0.0471	-0.0600	-0.0534
L_1	0.127	---	0.125	0.128
H_1	0.108	--	0.100	0.108
x_2	0.891	0.886	0.885	0.884
y_2	0.125	0.114	0.125	0.124
$\psi_2 (\times 10^4)$	6.62	6.44	6.43	6.45
ζ_2	-0.434	-0.394	-0.449	-0.455
L_2	0.262	---	0.265	0.264
H_2	0.320	-	0.320	0.324
x_4	0.004	--	---	-
y_4	0.004	-	-	-
$\psi_4 (\times 10^{10})$	-7.67	-	-	-
ζ_4	0.0009	--	-	-
L_4	0.004	---	---	---
H_4	0.004	---	---	---
x_5	0.992	-	0.990	0.992
y_5	0.008	--	0.010	0.008
$\psi_5 (\times 10^8)$	-1.86	-	-10.0	-7.76
ζ_5	0.0044	-	0.0092	0.0052
L_5	0.016	---	0.015	0.016
H_5	0.016	--	0.015	0.016

Table 3: Vortex locations, stream-function, and vorticity for $R_c = 400$.

	Present (coarse)	Present (fine)
x_0	0.524	0.522
y_0	0.548	0.556
ψ_0	-0.120	-0.134
ζ_0	2.17	2.57
x_1	0.089	0.087
y_1	0.097	0.096
$\psi_1 (\times 10^5)$	66.6	51.5
ζ_1	-0.679	-0.615
L_1	0.274	0.270
H_1	0.214	0.207
x_2	0.843	0.833
y_2	0.099	0.102
$\psi_2 (\times 10^4)$	24.3	24.7
ζ_2	-1.67	-1.65
L_2	0.338	0.360
H_2	0.395	0.403
x_3	0.033	0.032
y_3	0.884	0.884
$\psi_3 (\times 10^4)$	1.47	1.50
ζ_3	-0.806	-0.943
H_3^1	0.073	0.073
H_3^2	0.226	0.224
x_4	0.008	0.006
y_4	0.008	0.006
$\psi_4 (\times 10^8)$	-10.8	-2.83
ζ_4	0.0153	0.0056
L_4	0.012	0.012
H_4	0.012	0.012
x_5	0.992	0.991
y_5	0.008	0.009
$\psi_5 (\times 10^8)$	-25.6	-11.5
$\zeta_5 (\times 10^3)$	9.89	7.37
L_5	0.024	0.022
H_5	0.024	0.022

Table 4: Vortex locations, stream-function, and vorticity for $R_e = 2000$.

REPORT DOCUMENTATION PAGEForm Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 1997	3. REPORT TYPE AND DATES COVERED Technical Memorandum	
4. TITLE AND SUBTITLE Viscous Driven-Cavity Solver, User's Manual			5. FUNDING NUMBERS WU 242-80-01-01	
6. AUTHOR(S) William A. Wood				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-0001			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, D.C. 20546-0001			10. SPONSORING / MONITORING AGENCY REPORT NUMBER NASA TM-112874	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 2, 34			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The viscous driven-cavity problem is solved using a stream-function and vorticity formulation for the incompressible Navier-Stokes equations. This report provides the user's manual and FORTRAN code for the set of governing equations presented in NASA TM-110262.				
14. SUBJECT TERMS CFD; Incompressible; Navier-Stokes			15. NUMBER OF PAGES 51	
			16. PRICE CODE A04	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT	