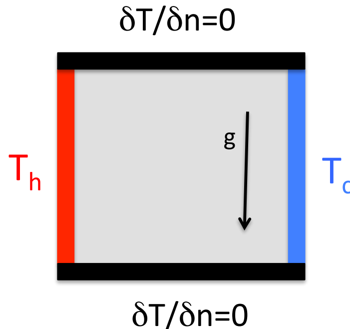
ME469

Computational Methods
in Fluid Mechanics

Handout #11
GI - W2015

Thermally Driven Cavity

The Problem



- Water in a heated cavity. What is the effect of gravity?
- What is the temperature of the bottom/top insulated walls? what is the heat flux on the vertical walls?

Thermally Driven Cavity

The Governing Equations (differential form)

Mass conservation

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0$$

Momentum balance

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = \frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} - \rho g_i$$

where $\tau_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \delta_{ij} \frac{\partial u_j}{\partial x_j}$

Energy conservation

$$\frac{\partial \rho e}{\partial t} + \frac{\partial \rho e u_j}{\partial x_j} = \frac{\partial u_i \tau_{ij}}{\partial x_j} - \frac{\partial u_j p}{\partial x_j} - \frac{\partial q_j}{\partial x_j}$$

where $e = \left(c_p T + \frac{1}{2} ||V||^2 + ||g|| \Delta z \right)$ and $q_j = -k \frac{\partial T}{\partial x_j}$

Assumptions already present:

- Newton-law for viscous stresses: τ_{ij}
- Fourier-law for heat flux: q_j

Additional assumptions:

- Ignore changes in potential energy: Δz small
- Neglect kinetic energy compared to internal energy $c_p T >> \frac{1}{2}||V||^2$: slow speed
- Neglect work done by the pressure and friction forces (viscous heating)
- Equation of state: $\rho = f(p, T) \approx \rho_{ref}$ = const
- Buoyancy force: $\rho g_i = \rho_{ref} \beta (T - T_{ref}) g_i$ (Boussinesq approximation: valid for small temperature differences)

Thermally Driven Cavity

The Governing Equations (Boussinesq approximation)

Mass conservation

$$\frac{\partial u_i}{\partial x_i} = 0$$

Momentum balance

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\frac{\mu}{\rho_{ref}}\left(\frac{\partial u_i}{\partial x_j}\right)\right] - \frac{1}{\rho_{ref}}\frac{\partial p}{\partial x_j} - \beta(T - T_{ref})g_i$$

Energy conservation

$$\frac{\partial T}{\partial t} + \frac{\partial u_j T}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\frac{k}{\rho_{ref} C_p}\left(\frac{\partial T}{\partial x_j}\right)\right]$$

Thermally Driven Cavity

Controlling Parameters, I

Fluid Properties

- Density: ρ_{ref}
- Thermal conductivity k
- Viscosity μ
- Specific heat C_p
- Thermal expansion coefficient β

Operating Conditions

- Wall temperatures: T_h, T_c
- Reference Temperature T_{ref}
- Gravity vector g_i

Non-dimensional Numbers

- Reynolds number: $Re = \frac{\rho U L}{\mu}$
- Grashof number: $Gr = \frac{\rho^2 ||g|| \beta (T_h - T_c) L^2}{\mu^2}$
- Prandtl number: $Pr = \frac{C_p \mu}{k}$
- Scaled temperature: $\theta = \frac{T - T_{ref}}{T_h - T_{ref}}$

Thermally Driven Cavity

The governing equations (non-dimensional Boussinesq)

Mass conservation

$$\frac{\partial u_i}{\partial x_i} = 0$$

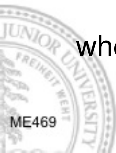
Momentum Balance

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} = \frac{1}{Re}\frac{\partial}{\partial x_j}\frac{\partial u_i}{\partial x_j} - \frac{\partial p}{\partial x_i} - \frac{Gr}{Re^2}\theta\tilde{g}_i$$

Energy Conservation

$$\frac{\partial \theta}{\partial t} + \frac{\partial \theta u_j}{\partial x_j} = \frac{1}{PrRe}\frac{\partial}{\partial x_j}\frac{\partial \theta}{\partial x_j}$$

where $\tilde{g}_i = \frac{g_i}{||g||}$



Computational Code

`mycavity.cpp`

- 2D, Navier Stokes solver for Boussinesq fluid
- Structured, colocated grid, implicit discretization
- SIMPLE fractional step
- Incomplete LU decomposition for solving linear system
- Visualization files in tecplot
- Written in C

Consider it a starting point...not a golden standard but a testbed for verification!

Solution Procedure

1. Start with T^n and u^n (divergence free)
2. Solve for an intermediate velocity (implicitly)

$$u_i^* - u_i^n = \frac{\Delta t}{\rho_{ref}} \left(H_i^* - \frac{\delta p^n}{\delta x_i} \right) - \beta (T^n - T_{ref}) g_i$$

3. Solve the Poisson equation for $\Delta p = p' = p^{n+1} - p^n$

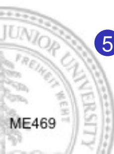
$$\frac{\delta}{\delta x_i} \left[\left(\frac{\Delta t}{\rho} \right) \frac{\delta p'}{\delta x_i} \right] = \frac{\delta u_i^*}{\delta x_i}$$

4. Correct the velocity field and update the pressure

$$u_i^{n+1} = u_i^* - \frac{1}{\rho_{ref}} \frac{\delta p'}{\delta x_i}; \quad p^{n+1} = p^n + p'$$

5. Solve for the temperature (implicitly)

$$T_i^{n+1} - T_i^n = \Delta t D_i^{n+1}$$



Solution Procedure

1. Start with T^n and u^n (divergence free)
2. Solve for an intermediate velocity (implicitly)

$$A_p^u u_P + \sum_{nb} A_{nb}^u u_{nb} = Q^u$$

3. Solve the Poisson equation for $\Delta p = p' = p^{n+1} - p^n$

$$A_P^p p'_P + \sum_{nb} A_{nb}^p p'_{nb} = Q^p = -\Delta \dot{m}_P^* - \Delta \tilde{\dot{m}}'_P$$

4. Correct the velocity field and update the pressure

$$u_e^{n+1} = u_e^* - \frac{S_e}{A_p^u}\left(p'_E - p'_P\right)$$

5. Solve for the temperature (implicitly)

$$A_p^T T_P + \sum_{nb} A_{nb}^T T_{nb} = Q^T$$

Solution Procedure

1. Start with T^n and u^n (divergence free)
2. Solve for an intermediate velocity (implicitly)

   ```
   calcuv()
   ```

3. Solve the Poisson equation for $\Delta p = p' = p^{n+1} - p^n$

   ```
   calcp()
   ```

4. Correct the velocity field and update the pressure

   ```
   correct()
   ```

5. Solve for the temperature (implicitly)

   ```
   calct()
   ```

Computational Code

mycavity.cpp

```
int main(int argc, char **argv){

    // read input file
    FILE *fp; fp=fopen("mycavity.in","r");
    [...]
    // allocate arrays
    x = new double[nx+1]; y = new double[ny+1];
    [...]
            // solve momentum equations
        resu = calcuv();

        // solve pressure equation and update momentum
        resp = calcp();

        // correct velocity and pressure field
        correct();

        // solve energy equation
        resT = calct();
    [...]
    // output global quantities of interest...
    output();
}
```

Computational Code

Controlling Parameters

Physical Properties

- cavity size: Lx=`lx`, Ly=`ly`
- integration time: `finaltime`
- gravity: g_i=[`gravx, gravy`]
- fluid properties: ρ_{ref}=`densit`, μ/ρ_{ref}=`visc`, β=`beta`, T_{ref}=`Tref`
- wall temperature: T_h=`Th`; T_c=`Tc`;

...mycavity.cpp

```
// read input file
FILE *fp; fp=fopen("mycavity.in","r");
fscanf(fp," %lf %lf %lf \n ",&lx,&ly,&finaltime); // domain size, number of inner steps
fscanf(fp," %lf %lf %lf \n",&densit,&visc,&prm); // density, viscosity, Pr #
fscanf(fp," %lf %lf %lf \n",&gravx,&gravy,&beta); // buoyancy force
fscanf(fp," %lf %lf %lf \n",&Th,&Tc,&Tref);  // wall BC
fscanf(fp," %d %d %lf %d %lf \n",&nx,&ny,&dt,&nsteps,&converged); // discretization
fscanf(fp," %d %d \n",&adim); // output
fclose(fp);
```

Discretization Properties

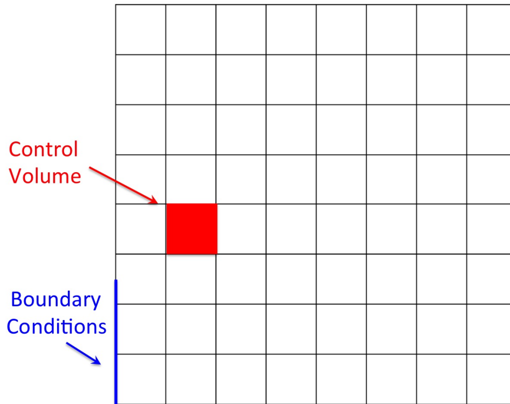
- grid size: [`nx`, `ny`]
- time step: `dt`
- inner iteration steps (non-linearity): `nsteps`
- ..few others are *hardcoded*

`...mycavity.cpp`

```
// read input file
FILE *fp; fp=fopen("mycavity.in","r");
fscanf(fp," %lf %lf %lf \n ",&lx,&ly,&finaltime); // domain size, number of inner steps
fscanf(fp," %lf %lf %lf \n",&densit,&visc,&prm); // density, viscosity, Pr #
fscanf(fp," %lf %lf %lf \n",&gravx,&gravy,&beta); // buoyancy force
fscanf(fp," %lf %lf %lf \n",&Th,&Tc,&Tref);  // wall BC
fscanf(fp," %d %d %lf %d %lf \n",&nx,&ny,&dt,&nsteps,&converged); // discretization
fscanf(fp," %d %d \n",&adim); // output
fclose(fp);
```

Computational Grid

Vertices

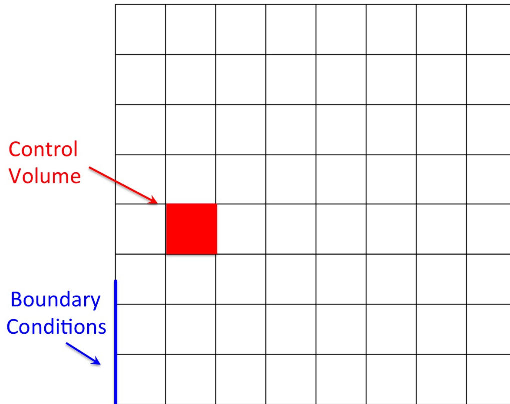


- Indices: `i`, `j`
- Global CV index: `ij`
- Vertex coordinates: `x[i]`, `y[j]` (Cartesian grid)
- CV centers coordinates: `xc[i]`; `yc[j]`
- Unknowns in CV: `u[ij]`, `v[ij]`, `p[ij]`, `T[ij]`
- Time-history:
  - `u0`=u^{n+1}, `u0`=u^{n}, `u00`=u^{n-1}
  - similarly for other variables..
  - no-time history for `p`. But we need Δp=`pp`
- Implicit operator:
  - left hand side: `ap[ij]`, `aw[ij]`, `ae[ij]`, `an[ij]`, `as[ij]`
  - right hand side: `su[ij]`, `sv[ij]`

Heavy reuse of memory in the code (e.g. u^* is not stored)

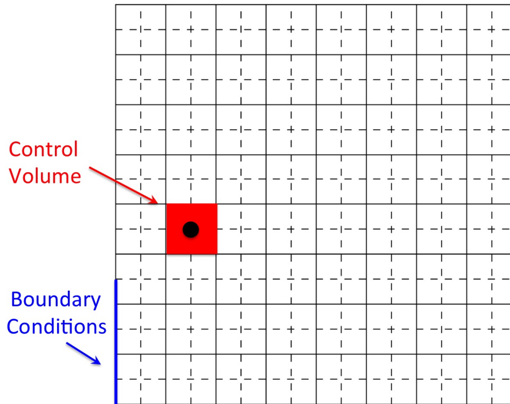
Computational Grid

Understanding the indices



Computational Grid

CVs



Computational Grid

Indices - NX=NY=10

(NX,NY)

(1,NY-1)

(NX-1,NY-1)

unknowns

(2,2)

(NX-1,2)

(NX,2)

i

j

Computational Grid

Comments

- The Computational grid is generated in routine `grid()`
- Ordering
  - The indices (`i`, `j`) are *notionally* connected to CVs centroids
  - The cavity boundaries correspond to `i=1`, `i=nx`, `j=1` and `j=ny`
  - The CVs are in the range `i` $\in$ [`2`, `nx-1`] and `j` $\in$ [`2`, `ny-1`]
- Global index
  - An *offset* index array is introduced: `li[i]=(i-1)*ny`
  - The `ij` index is defined as `ij=j+li[i]`
  - The complete `ij` range is `ij` $\in$ [`1`, `nx*ny`] and include CV centroids and boundaries
- In addition to vertices & CV centroids coordinates it also stores the weight factor (face-centroid location): `fx[i]=(x[i]-xc[i])/(xc[i+1]-xc[i]); fy[j]...`

Loop over all CVs

```
for (int i=2; i<=nx-1; i++) {
  for (int j=2; j<=ny-1; j++) {
    int ij = li[i]+j;
    u[ij]=...
    ...
```

Loop over *south* faces of CVs

```
// south boundaries
for (int i=2; i<=nx-1; i++) {
  int ij=li[i]+1;
  u[ij]=...
  ...
```

Loop over *west* faces of CVs

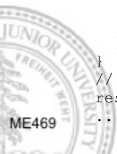
```
// west boundaries
for (int j=2; j<=ny-1;j++) {
  int ij=li[2]+j;
  u[ij]=...
  ...
```

Solution Procedure

Routine `calcuv`

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
double calcuv() { // solve the momentum equations
  int nsw = 5; // iterations for the linear system solver
  double tol = 0.2; // tolerance for the linear system solver
  double urf = 0.8; // under-relaxation factor

  uvlhs(); // computing fluxes...
  uvrhs(); // computing source terms...
  // time terms (gamt 0=backward implicit; non-zero=3-level scheme)
  for (int i=2; i<=nx-1; i++) {
    for (int j=2; j<=ny-1; j++) {
      int ij = li[i]+j;
      double dx = x[i]-x[i-1];
      ...
      apv[ij]=apv[ij]+(1.+0.5*gamt)*apt;
    }
  }
  uvbc(); // apply boundary conditions
  // apply under-relaxation for u
  for (int i=2; i<=nx-1; i++) {
    for (int ij=li[i]+2;ij<=li[i]+ny-1;ij++) {
      ...
      ap[ij] = (apu[ij]-ae[ij]-aw[ij]-an[ij]-as[ij])/urf;
      su[ij] = su[ij] +(1.-urf)*ap[ij]*u[ij];
      apu[ij]=1./ap[ij];
    }
  }
  // solve linear system for u...
  resu = sipsol(nsw, u, su, tol);
  ...
```



Solution Procedure

Routine `calcuv()`

- Evaluate left hand side (implicit flux operator): `uvlhs()`
- Evaluate right hand side (explicit flux operator and buoyancy terms): `uvrhs()`
- Apply boundary conditions: `uvbc()`
- Evaluate time term and add to left or right hand side
- Apply under-relaxation factor: `urf`
- Solve linear systems
  - `resu = sipsol(nsw, u, su, tol);`
  - `resv = sipsol(nsw, v, sv, tol);`
- *Hardcoded* parameters: `urf=0.9`, `nsw=5` and `tol=0.2`

Solution Procedure

Routine `resu = sipsol(nsw, u, su, tol)`

- SIP: Strong Implicit Procedure (Stone's Implicit Procedure)
- Incomplete LU solver - iterative linear system solver
- Stops after `nsw` steps of if the residual `resu` $\leq$ `too`

SIAM J. NUMER. ANAL.
Vol. 5, No. 3, September 1968
Printed in U.S.A.

ITERATIVE SOLUTION OF IMPLICIT APPROXIMATIONS OF MULTIDIMENSIONAL PARTIAL DIFFERENTIAL EQUATIONS*

HERBERT L. STONE†

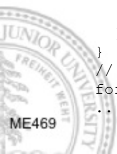
Summary. A new iterative method has been developed for solving the large sets of algebraic equations that arise in the approximate solution of multidimensional partial differential equations by implicit numerical techniques. This method has several advantages over those now in use. First, its rate of convergence does not depend strongly on the nature of the coefficient matrix of the equations to be

- Can be replaced easily by Jacobi, Gauss-Seidel, Conjugate-gradient-like approaches, Krylov solvers, etc..

Solution Procedure

Routine `uvlhs`

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void uvlhs() { // implicit flux discretization for the momentum equations
  // east/west fluxes
  for (int i=2; i<=nx-2;i++) {
    double fxe =fx[i];
    double fxp =1.-fxe;
    double dxpe=xc[i+1]-xc[i];
    for (int j=2; j<=ny-1;j++) {
      int ij=li[i]+j;
      int ije=ij+ny;
      double s=(y[j]-y[j-1]);
      double d=visc*s/dxpe;
      double ce=fmin(f1[ij],0.);
      double cp=fmax(f1[ij],0.);
      double fuuds=cp*u[ij]+ce*u[ije];
      double fvuds=cp*v[ij]+ce*v[ije];
      double fucds=f1[ij]*(u[ije]*fxe+u[ij]*fxp);
      double fvcds=f1[ij]*(v[ije]*fxe+v[ij]*fxp);
      ae[ij] = ce-d;
      aw[ije]=-cp-d;
      su[ij] =su[ij] +gds*(fuuds-fucds);
      su[ije]=su[ije]-gds*(fuuds-fucds);
      sv[ij] =sv[ij] +gds*(fvuds-fvcds);
      sv[ije]=sv[ije]-gds*(fvuds-fvcds);
    }
  }
  //  south/north fluxes
  for (int j=2; j<=ny-2;j++) {
  ...
```



Solution Procedure

Routines `uvlhs`

- `uvlhs` computes fluxes, i.e. loop on faces and stores quantities for the CVs at either side of the face (avoid double work)
- The convective fluxes are computed using both *upwind* and *central* differencing stencils - the parameter `gds` blends the two (hardcoded to `gds=0.9`)
- mass flux through faces is stored in separate arrays `f1` and `f2`: this is used to enforce mass conservation for the CV
- Each momentun equation has an RHS array (`su`, `sv`) but *only* one LHS matrix is stored!

Solution Procedure

Routine `uvrhs`

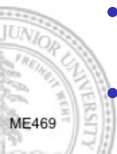
```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void uvrhs() { // source terms for the momentum equations

  for (int i=2; i<=nx-1;i++) {
    double dx=x[i]-x[i-1];
    for (int j=2; j<=ny-1;j++) {
      double dy=y[j]-y[j-1];
      double vol=dx*dy;
      int ij=li[i]+j;
      double pe=p[ij+ny]*fx[i]+p[ij]*(1.-fx[i]);
      double pw=p[ij]*fx[i-1]+p[ij-ny]*(1.-fx[i-1]);
      double pn=p[ij+1]*fy[j]+p[ij]*(1.-fy[j]);
      double ps=p[ij]*fy[j-1]+p[ij-1]*(1.-fy[j-1]);
      dpx[ij]=(pe-pw)/dx;
      dpy[ij]=(pn-ps)/dy;
      su[ij]=su[ij]-dpx[ij]*vol;
      sv[ij]=sv[ij]-dpy[ij]*vol;
      double sb=-beta*densit*vol*(T[ij]-Tref);
      su[ij]=su[ij]+gravx*sb;
      sv[ij]=sv[ij]+gravy*sb;
      }
   }
}
```

- Each discretized equation looks like:

$$A_p\phi_P + \sum_{nb} A_{nb}\phi_{nb} = Q$$

- We do not solve for ϕ directly but under-relax the update to achieve better convergence: $\phi^{new} = \phi^{old} + \alpha\Delta\phi$
- How is this implemented?
- Explicitly
  1. solve $A_p\phi_p + \sum_{nb} A_{nb}\phi_{nb} = Q$
  2. relax $\phi^{new} = \alpha\phi + (1-\alpha)\phi^{old}$
- Implicitly:
  1. solve $A_p\phi_p/\alpha + \sum_{nb} A_{nb}\phi_{nb} = Q + A_p\phi^{old}(1-\alpha)/\alpha$
- The choice of α (`urf`) is not straightforward.



Solution Procedure

Correction step: `correct()`

- Solution Step:

$$A_p^u u_P + \sum_{nb} A_{nb}^u u_{nb} = Q^u$$

$$A_P^p p'_P + \sum_{nb} A_{nb}^p p'_{nb} = Q^p$$

- Correction Step:

$$u_e^{n+1} = u_e^* - \frac{S_e}{A_p^u}\left(p'_E - p'_P\right)$$

$$p^{n+1} = p^n - + p'$$

Solution Procedure

Routine `correct()`

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void correct() { // correct velocity and pressure field

  double urf = 0.2; // under-relaxation factor for pressure

  // set reference pp
  int ijpref=li[3]+3; // a "random" internal point...
  double ppo=pp[ijpref];

  //  correct mass fluxes
  for (int i=2; i<=nx-2;i++) {
    for (int ij=li[i]+2;ij<=li[i]+ny-1;ij++) {
      f1[ij]=f1[ij]+ae[ij]*(pp[ij+ny]-pp[ij]);
    }
  }
  ...
  // correct cell center velocity and pressure
  for (int i=2; i<=nx-1;i++) {
    double dx=x[i]-x[i-1];
    for (int j=2; j<=ny-1;j++) {
      int ij=li[i]+j;
      double dy=y[j]-y[j-1];
      double ppe=pp[ij+ny]*fx[i]+pp[ij]*(1.-fx[i]);
      ...
      u[ij]=u[ij]-(ppe-ppw)*dy*apu[ij];
      v[ij]=v[ij]-(ppn-pps)*dx*apv[ij];
      p[ij]=p[ij]+urf*(pp[ij]-ppo);
    }
  }
}
```

Solution Procedure

Routine `calcp()`

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
double calcp() { // solve the pressure equation and update momentum

  int nsw = 200; // iterations for the linear system solver
  double tol = 0.02; // tolerance for the linear system solver

  //  east and west fluxes and coefficients
  for (int i=2; i<=nx-2;i++) {
    double dxpe=xc[i+1]-xc[i];
    double fxe=fx[i];
    double fxp=1.-fxe;
    for (int j=2;j<=ny-1;j++) {
      int ij=li[i]+j;
      int ije=ij+ny;
      double s=(y[j]-y[j-1]);
      double vole=dxpe*s;
      double d=densit*s;
      double dpxel=0.5*(dpx[ije]+dpx[ij]);
      double uel=u[ije]*fxe+u[ij]*fxp;
      double apue=apu[ije]*fxe+apu[ij]*fxp;
      double dpxe=(p[ije]-p[ij])/dxpe;
      double ue=uel-apue*vole*(dpxe-dpxel);
      f1[ij]=d*ue;
      ae[ij]=-d*apue*s;
      aw[ije]=ae[ij];
    }
  }
```

Solution Procedure

Routine `calcp()`

- What are: `ae`, `apu`, `apue`
- Construction of the pressure correction equation

$$\texttt{ae[ij]} = A_E^p = -\left(\frac{\rho S^2}{A_P^u}\right)_e = -\left(\rho S^2\right)_e * \texttt{apue[ij]}$$

- Rhie-Chow Interpolation:

$$u_e = \overline{u_e} + \frac{1}{\overline{A_e^u}}\left(\overline{\left(\frac{\delta p}{\delta x}\right)_e} - \left(\frac{\delta p}{\delta x}\right)_e\right)$$

$$\texttt{f1[ij]} = (\rho S)_e * u_e$$

Solution Procedure

Final comments

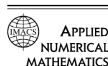
- Mass conservation is enforced through the *face* velocities: `f1[]`, `f2[]` (mass-conserving velocities)
- Pressure-velocity decoupling is eliminated using the Rhie-Chow interpolation for the mass-conserving velocities
- Convective fluxes (in momentum AND energy transport) are based on the mass conserving velocities
- Only pressure gradients are important. Pressure is *anchored*...

Thermally Driven Cavity

- This is a *classical* test case for Navier-Stokes
- Many references available...

ELSEVIER

Applied Numerical Mathematics 40 (2002) 327–336



www.elsevier.com/locate/apnum

Thermally driven cavity flow with Neumann condition for the pressure

Obidio Rubio [a,*], Elba Bravo [b], Julio R. Claeyssen [c]

[a] *Facultad de Ciencias, Universidad Nacional de Trujillo, Trujillo, Peru*
[b] *Departamento de Engenharia e Computação, Universidade Regional Integrada-URI, 98.400-000 Frederico Westphalen, RS, Brazil*
[c] *Universidade Federal do Rio Grande do Sul, IM-PROMEC, P.O. Box 10673, 90.001-000 Porto Alegre, RS, Brazil*

Abstract

We develop a velocity-pressure algorithm with a pressure Neumann condition in primitive variables using finite differences, for a 2D thermally driven square cavity flow with the Boussinesq approximation and a fixed Prandtl number. The pressure field is updated in a one-step weighted form. Simulations were made for several Rayleigh numbers and the results are close to those found in the literature. © 2002 IMACS. Published bv Elsevier Science