|  |
| --- |
|  |
| ME 811: HW 5 |
|  |
|  |
| **Michael Crawley** |
|  |

|  |
| --- |
|  |

# Part I

### CONDIF

The hybrid difference scheme has been widely used in computational fluid dynamics due to several beneficial attributes it possesses, namely that it is stable, does not produce numerical oscillations, remains bounded for all Peclet numbers and is relatively easy to implement. However, a major drawback in using hybrid differencing exists: it produces high levels of false numerical diffusion for high Peclet numbers and nonzero flow angles, which makes it ill-suited for some applications, particularly recirculating flows. While there are a number of higher accuracy schemes that do not produce unacceptable numerical diffusion as the hybrid scheme does, they too possess significant disadvantages such as over/under shoots, issues with stability, higher computational costs and increased complexity of implementation. The second order central differencing scheme, unlike the hybrid scheme, is unstable for all but the lowest Peclet number flows. However, it is enticing due to its second order accuracy, extreme ease of implementation, and lack of numerical diffusion. It is with this in mind that the CONDIF scheme was developed.

The central theme of CONDIF is to modify the central difference scheme such that its lack of numerical diffusion is retained while ensuring that the coefficient matrices are all positive, and hence the scheme is numerically stable. To do this, the governing equation is transformed so that the scalar variable is expressed in a ratio involving the scalar gradients at the cell faces. The structure of the differencing scheme at each cell is then defined based upon both the Peclet number, and this ratio. For low Peclet numbers (Pe < 2), for which the central differencing scheme is stable, the central differencing scheme is retained. If the Peclet number is greater than 2, but the gradient ratio is positive (meaning that the scalar value is monotonically increasing or decreasing in the node), the central differencing scheme is again used. When the Peclet number is greater than 2 and the gradient ratio is negative however, the upwind scheme is utilized instead in order to ensure numerical stability and boundedness.

Simulations comparing the CONDIF scheme against the hybrid scheme have found that CONDIF has, at the very worst, similar accuracy as the hybrid scheme. For high Peclet number flows, CONDIF results in significantly lower numerical error, particularly for coarse meshes. At low Peclet numbers, the CONDIF and hybrid schemes have nearly identical errors, which is to be expected as they both reduce to the central differencing scheme for this flow condition. Like the hybrid scheme, CONDIF does not exhibit issues with numerical stability. Unlike the hybrid scheme however, CONDIF does not produce significant levels of numerical diffusion near high gradients, even when the flow angle is increased. Based on these results, the CONDIF scheme appears to be a viable alternative to the hybrid scheme for many flows, as it is numerically stable, accurate, and does not introduce significant levels of numerical diffusion. Fortunately, in most cases it was found that the CONDIF scheme did not require significantly higher computational time over comparable hybrid scheme simulations. CONDIF has the additional benefit of being simple to implement, particularly in low dimensional flows where the researcher is interest in a scalar value. CONDIF can be extended to fully three dimensional flows and vector variables, though its implementation becomes significantly more complex.

### SMART

Standard high order finite difference schemes based on Taylor series expansions such as QUICK have been shown to resolve smooth functions well. However, when high gradients are introduced into the function, methods such as QUICK produce high frequency errors in the solution due to the fact that the truncation error associated with the Taylor series expansion is of the same order as the function itself. Furthermore, as the truncation error is on the same order as the function for all reasonable expansion orders, increasing the order of the differencing scheme does not inhibit this behavior. Schemes such as QUICK and CUI have the additional drawback of being convectively unbounded and, though they are upwind biased, the associated coefficient matrices are numerically unstable for certain situations where the matrices become non-diagonally dominant. Hence, traditional schemes are unable to concurrently provide high accuracy, low numerical diffusion, numerical stability and boundedness. The SMART algorithm was proposed in order to rectify these problems.

The SMART algorithm was developed using the curvature-compensated convective transport method, where a tuning parameter is introduced into the standard upwind biased differencing scheme for the convective terms based upon the upstream curvature of the function being approximated. This parameter is used to produce an optimized upstream weighting for the convection terms. The parameter is varied so as to produce a convectively stable, bounded solution at all nodes, with high accuracy. Unfortunately, ensuring boundedness requires forgoing high accuracy of the solution. As the value of the tuning parameter is based upon the value of the function, and iterative solution process must be used where values of the function at previous iterations must be used to calculate an intermediate iteration function value, from which the value of the tuning parameter can then be calculated. Finally, the function value at the new iteration can then be calculated based on the previous iteration and the tuning parameter.

Simulations comparing results from the SMART algorithm against those calculated using the QUICK and upwind schemes for resolution of step functions have shown that the SMART does not produce the undesirable qualities of the other schemes namely significant artificial diffusion in the case of the upwind scheme, or over/under shoots in the case of the QUICK scheme. The SMART solution is both more accurate than the QUICK or upwind solutions, and it remains bounded. There is a cost associated with the SMART scheme however, in the form of increased computational time required due to the intermediate iteration steps required to calculate the tuning parameter. However, the SMART algorithm is able to produce identical levels of accuracy to the QUICK or upwind schemes for problems involve high gradients with far fewer nodes and hence actual computational time can be reduced via implementation of SMART. Implementation of the SMART algorithm is far more intensive than QUICK, and is not justified when it is known that the QUICK method will produce acceptable solutions. Additionally, in some cases SMART may result in oscillations in the iterations that reduce its convergence. Hence, SMART should probably only be used in situations where it is known or suspected that the solution will involve steep gradients that will lead to unbounded solutions by the QUICK method.

### CUBISTA

As with the SMART algorithm, CUBISTA was developed in order to ensure bounded, non-oscillatory solutions in regions of high gradients, without introducing excessive numerical diffusion. CUBISTA, however, has and additional focus on numerical convergence, which has been found to be diminished in other high resolution schemes such as SMART. In CUBISTA, this is accomplished via satisfying the total variation diminishing (TVD) constraints. As with the SMART scheme, the CUBISTA scheme is formed by modifying the QUICK scheme for the convective flux terms by the introduction of a tuning parameter. This parameter is found by numerical experimentation, but must satisfy the TVD requirements. Generally, a value of 0.25 is used for the parameter, as this value provides a good compromise between resolution of high gradient regions and numerical convergence.

Simulations of step profiles comparing results from the CUBISTA and SMART algorithms have found that CUBISTA results in a slightly higher level of numerical diffusion and a reduced order of accuracy (2.7 versus 3 for the QUICK scheme upon which it is based). Surprisingly, CUBISTA results in a lower level of asymmetry for symmetric profiles than that produced by the SMART algorithm. Most importantly, CUBISTA was shown to be able to converge to a prescribed tolerance for a given flow configuration, whereas the SMART algorithm failed to do so with the given time increments. The SMART algorithm required significantly smaller time steps and hence, required significantly more iterations than CUBISTA in order to converge to the steady state solution. These results suggest that CUBISTA should be used in situations where high accuracy and low numerical diffusion are required, but the SMART algorithm may have difficulty converging (viscoelastic fluid flows are one situation mentioned in the literature). CUBISTA does not require any additional complexities in implementation over that of the SMART algorithm, though it is still far more intensive than the QUICK method.

# Part II

Figure 1 shows the pressure contours and velocity vectors for a Reynolds number flow with 40 by 40 and 80 by 80 meshes, computed via the SIMPLE and SIMPLEC algorithms. Based on these plots, it is concluded that the solutions calculated using the two algorithms yield identical results. This is not surprising, as the modifications to the SIMPLE algorithm to produce the SIMPLEC algorithm relate to the correction terms, which go to zero as convergence is reached. A similar result can be seen in Figure 2, where the centerline u and v velocities have been plotted for Reynolds numbers 100 and 500 and meshes of 40 by 40 and 80 by 80, computed via SIMPLE and SIMPLEC. Again, the velocity contours are identical between the two algorithms. For brevity, results for other cases have been omitted, as no new information can be gleaned from them. Figure 3 and Figure 4 show the velocity and pressure residuals from the two algorithms for meshes of 40 by 40 and 80 by 80, respectively. For a given linear and inertial relaxation factor, the number of iterations required for SIMPLE and SIMPLEC to converge are nearly identical, with the SIMPLEC method converging a few iterations faster than the SIMPLE. Surprisingly, even the oscillatory behavior of the residuals is similar between the two algorithms for a given condition. It was found that the SIMPLEC algorithm failed to converge for certain conditions and relaxation factors (in this case, Reynolds number of 1000, alpha of 0.2 and omega of 0.5), for which the SIMPLE algorithm converged without issue (not shown). This suggests that though the SIMPLEC algorithm has fewer tuning parameters than the SIMPLE algorithm, SIMPLEC is much more sensitive to the choice of relaxation factors.

|  |  |
| --- | --- |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\40x40 mesh, Re = 500 pressure.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\40x40 Re500\40x40 mesh, Re = 500 pressure.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\80x80 mesh, Re = 500 pressure.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\80x80 Re500\80x80 mesh, Re = 500 pressure.png |

Figure 1: Pressure contours and velocity vectors for Reynolds number of 500, for 40 by 40 mesh (top) 80 by 80 mesh (bottom), SIMPLE (left) and SIMPLEC (right).

|  |  |
| --- | --- |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\40x40 mesh, Re = 500 x centerline.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\40x40 Re500\40x40 mesh, Re = 500 x centerline.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\40x40 mesh, Re = 500 y centerline.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\40x40 Re500\40x40 mesh, Re = 500 y centerline.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\80x80 mesh, Re = 500 x centerline.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\80x80 Re500\80x80 mesh, Re = 500 x centerline.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\80x80 mesh, Re = 500 y centerline.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\80x80 Re500\80x80 mesh, Re = 500 y centerline.png |

Figure 2: Velocity contours for Reynolds number 500, 40 by 40 and 80 by 80 meshes, for SIMPLE (left) and SIMPLEC (right).

|  |  |
| --- | --- |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\40x40 mesh, Re = 100 Residuals.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\40x40 Re100\40x40 mesh, Re = 100 Residuals.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\40x40 mesh, Re = 500 Residuals.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\40x40 Re500\40x40 mesh, Re = 500 Residuals.png |

Figure 3: Velocity and pressure residuals for 40 by 40 mesh, for Reynolds number of 100 (top) and Reynolds number of (500) bottom, SIMPLE (left) and SIMPLEC (right).

|  |  |
| --- | --- |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\80x80 mesh, Re = 100 Residuals.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\80x80 Re100\80x80 mesh, Re = 100 Residuals.png |
| Z:\My Documents\Matlab\Coursework\ME 811\HW 4\80x80 mesh, Re = 500 Residuals.png | Z:\My Documents\Matlab\Coursework\ME 811\HW 5\80x80 Re500\80x80 mesh, Re = 500 Residuals.png |

Figure : Velocity and pressure residuals for 80 by 80 mesh, for Reynolds number of 100 (top) and Reynolds number of (500) bottom, SIMPLE (left) and SIMPLEC (right).

## Appendix

function [u v p Resx Resy] = SIMPLEC(rho,gamma,alpha,omega,Ulid,M,N)

tic

Rtol = 10E-10;

itrmax = 1E4;

L = 0.01;

dx = L/N;

dy = L/M;

x = 0:dx:L;

y = 0:dy:L;

xc = dx/2:dx:L-dx/2;

yc = dy/2:dy:L-dy/2;

%initialize variables

u = zeros(M,N);

v = u; p = u;

% v = ones(M,N); p = v; v(end,:) = 0; u = ones(M,N); u(:,1) = 0;

Resx = zeros(1,itrmax+1);Resx(1) = 1;

counter = 1;

Resy = Resx; Resp = Resx;

while (Resx(counter) >= Rtol || Resy(counter) >= Rtol || Resp(counter) >= Rtol) && counter <= itrmax

[AxO AxE AxW AxN AxS Sx] = calcXcoefs(u,v,p,Ulid,rho,gamma,dx,dy);

[AyO AyE AyW AyN AyS Sy] = calcYcoefs(u,v,p,rho,gamma,dx,dy);

[uh Resx(counter+1)] = Xsolver(alpha,AxO,AxE,AxW,AxN,AxS,Sx,u);

uh = u+uh;

[vh Resy(counter+1)] = Ysolver(alpha,AyO,AyE,AyW,AyN,AyS,Sy,v);

vh = v+vh;

[ApO ApE ApW ApN ApS Sp] = calcPcoefs(alpha,rho,dx,dy,uh,vh,AxO,AxE,AxW,AxN,AxS,AyO,AyE,AyW,AyN,AyS);

[pp Resp(counter+1)] = Psolver(ApO,ApE,ApW,ApN,ApS,Sp);

u(:,2:end) = uh(:,2:end)+omega\*dy\*(pp(:,1:end-1)-pp(:,2:end))./((1+alpha)\*AxO+AxE+AxW+AxN+AxS);

v(1:end-1,:) = vh(1:end-1,:)+omega\*dx\*(pp(2:end,:)-pp(1:end-1,:))./((1+alpha)\*AyO+AyE+AyW+AyN+AyS);

p = p+pp;

fprintf('counter: %i Resx: %1.2e Resy: %1.2e Resp: %1.2e \n',[counter Resx(counter+1) Resy(counter+1) Resp(counter+1)]);

counter = counter+1;

end

u = [u zeros(M,1)];

u = flipud(u);

v = [zeros(1,N); v];

v = flipud(v);

p = flipud(p);

Resx = Resx(2:counter);

Resy = Resy(2:counter);

Resp = Resp(2:counter);

foldername = [num2str(M),'x',num2str(N),' Re',num2str(round(rho\*Ulid\*L/gamma))];

if exist(foldername,'file') ~= 7

mkdir(pwd,foldername);

end

cd(foldername);

compute\_time = toc;

save data.mat;

filename = [num2str(M),'x',num2str(N),' mesh, Re = ',num2str(round(rho\*Ulid\*L/gamma))];

%Plot centerline u,v velocity along centerlines

h(1) = figure; %y = const figure

plot(x,(u(M/2,:)+u(M/2+1,:))/2,'k',xc,v(M/2+1,:),'k--');legend('u velocity','v velocity','Location','Best');xlabel('x');ylabel('(m/s)');title(['Horizontal Centerline Velocities for ',filename]);

h(2) = figure; %x = const figure

plot(yc,u(:,N/2+1),'k',y,(v(:,N/2)+v(:,N/2+1))/2,'k--');legend('u velocity','v velocity','Location','Best');xlabel('y');ylabel('(m/s)');title(['Vertical Centerline Velocities for ',filename]);

%Plot residuals

h(3) = figure;

semilogy(1:counter-1,Resx,'k',1:counter-1,Resy,'k--',1:counter-1,Resp,'k-.');legend('u Residual','v Residual','p Residual');xlabel('iteration');ylabel('Residual');title(['Outer Iteration Residuals for ',filename]);

%Plot quiver and pressure

uc = (u(:,1:end-1)+u(:,2:end))/2;

vc = (v(1:end-1,:)+v(2:end,:))/2;

h(4) = figure;

contourf(xc,yc,p,50);colormap(flipud(gray));colorbar;hold on;

quiver(xc,yc,uc,vc,3);xlabel('x');ylabel('y');hold off;

title(['Velocity Vectors and Pressure Contours for ',filename]);

%Save figures

saveas(h(1),[filename,' y centerline'],'fig');saveas(h(1),[filename,' y centerline'],'png');

saveas(h(2),[filename,' x centerline'],'fig');saveas(h(2),[filename,' x centerline'],'png');

saveas(h(3),[filename,' Residuals'],'fig');saveas(h(3),[filename,' Residuals'],'png');

saveas(h(4),[filename,' pressure'],'fig');saveas(h(4),[filename,' pressure'],'png');

close(h);

cd ..

end

function [AxO AxE AxW AxN AxS Sx] = calcXcoefs(u,v,p,Ulid,rho,gamma,dx,dy)

%Calculates link coefficients for x-momentum FVM equation

%Note: for an NxMy mesh, u(:,1) is zero while v(:,1) and p(:,1) are not;

%the u and v velocity fields are modified to reflect this.

%Inputs:

% u: x-velocity component (NxM matrix)

% v: y-velocity component (NxM matrix)

% p: pressure (NxM matrix)

% Ulid: lid velocity

% rho: density (constant)

% gamma: viscosity (constant)

% dx: step size in x-direction

% dy: step size in y-direction

[M N] = size(u);

ut = [u zeros(M,1)]; %add known boundary values to simplify computations

vt = [zeros(1,N); v]; %add known boundary values to simplify computations

ce = rho\*0.5\*(ut(:,2:end-1)+ut(:,3:end));

cw = rho\*0.5\*(ut(:,2:end-1)+ut(:,1:end-2));

cn = rho\*0.5\*(vt(1:end-1,2:end)+vt(1:end-1,1:end-1));

cs = rho\*0.5\*(vt(2:end,2:end)+vt(2:end,1:end-1));

dep = 0.5\*(abs(ce)+ce);

dem = 0.5\*(abs(ce)-ce);

dwp = 0.5\*(abs(cw)+cw);

dwm = 0.5\*(abs(cw)-cw);

dsp = 0.5\*(abs(cs)+cs);

dsm = 0.5\*(abs(cs)-cs);

dnp = 0.5\*(abs(cn)+cn);

dnm = 0.5\*(abs(cn)-cn);

%Calculate link coefficients for interior nodes

AxO = (dep+dwm+2\*gamma/dx)\*dy+(dnp+dsm+2\*gamma/dy)\*dx;

AxE = -(dem+gamma/dx)\*dy;

AxW = -(dwp+gamma/dx)\*dy;

AxN = -(dnm+gamma/dy)\*dx;

AxS = -(dsp+gamma/dy)\*dx;

Sx = - (p(:,2:end)-p(:,1:end-1))\*dy;

%Modify link coefficients for North Boundary

AxO(1,:) = (dep(1,:)+dwm(1,:)+2\*gamma/dx)\*dy+(dnp(1,:)+4\*gamma/dy)\*dx;

AxN(1,:) = 0;

AxS(1,:) = -(dsp(1,:)+4\*gamma/dy/3)\*dx;

Sx(1,:) = Sx(1,:)+8\*gamma\*dx\*Ulid/3/dy;

%Modify link coefficients for South Boundary

AxO(M,:) = (dep(M,:)+dwm(M,:)+2\*gamma/dx)\*dy+(dnp(M,:)+4\*gamma/dy)\*dx;

AxS(M,:) = 0;

AxN(M,:) = -(dnm(M,:)+4\*gamma/dy/3)\*dx;

%Modify link coefficients for West Boundary

AxW(:,1) = 0;

%Modify link coefficients for East Boundary

AxE(:,end) = 0;

end

function [AyO AyE AyW AyN AyS Sy] = calcYcoefs(u,v,p,rho,gamma,dx,dy)

%Calculates link coefficients for y-momentum FVM equation

%Inputs:

% u: x-velocity component (NxM matrix)

% v: y-velocity component (NxM matrix)

% p: pressure (NxM matrix)

% Ulid: lid velocity

% rho: density (constant)

% gamma: viscosity (constant)

% dx: step size in x-direction

% dy: step size in y-direction

[M N] = size(u);

ut = [u zeros(M,1)]; %add known boundary values to simplify computations

vt = [zeros(1,N); v]; %add known boundary values to simplify computations

ce = rho\*0.5\*(ut(1:end-1,2:end)+ut(2:end,2:end));

cw = rho\*0.5\*(ut(1:end-1,1:end-1)+ut(2:end,1:end-1));

cn = rho\*0.5\*(vt(2:end-1,:)+vt(1:end-2,:));

cs = rho\*0.5\*( vt(2:end-1,:)+vt(3:end,:));

dep = 0.5\*(abs(ce)+ce);

dem = 0.5\*(abs(ce)-ce);

dwp = 0.5\*(abs(cw)+cw);

dwm = 0.5\*(abs(cw)-cw);

dsp = 0.5\*(abs(cs)+cs);

dsm = 0.5\*(abs(cs)-cs);

dnp = 0.5\*(abs(cn)+cn);

dnm = 0.5\*(abs(cn)-cn);

%Calculate link coefficients for interior nodes

AyO = (dep+dwm+2\*gamma/dx)\*dy+(dnp+dsm+2\*gamma/dy)\*dx;

AyE = -(dem+gamma/dx)\*dy;

AyW = -(dwp+gamma/dx)\*dy;

AyN = -(dnm+gamma/dy)\*dx;

AyS = -(dsp+gamma/dy)\*dx;

Sy = -(p(1:end-1,:)-p(2:end,:))\*dx;

%Modify link coefficients for North Boundary

AyN(1,:) = 0;

%Modify link coefficients for South Boundary - unnecessary

AyS(M-1,:) = 0;

%Modify link coefficients for East Boundary

AyO(:,N) = (dwm(:,N)+4\*gamma/dx)\*dy+(dnp(:,N)+dsm(:,N)+2\*gamma/dy)\*dx;

AyE(:,N) = 0;

AyW(:,N) = -(dwp(:,N)+4\*gamma/dx/3)\*dy;

%Modify link coefficients for West Boundary

AyO(:,1) = (dep(:,1)+4\*gamma/dx)\*dy+(dnp(:,1)+dsm(:,1)+2\*gamma/dy)\*dx;

AyE(:,1) = -(dem(:,1)+4\*gamma/dx/3)\*dy;

AyW(:,1) = 0;

end

function [ApO ApE ApW ApN ApS Sp] = calcPcoefs(alpha,rho,dx,dy,uh,vh,AxO,AxE,AxW,AxN,AxS,AyO,AyE,AyW,AyN,AyS)

[M,~] = size(AxO);

[~,N] = size(AyO);

sAx = [Inf\*ones(M,1) ((1+alpha)\*AxO+AxE+AxW+AxN+AxS) Inf\*ones(M,1)]; %AxO = (:,1:end-1), AxE = (:,2:end)

sAy = [Inf\*ones(1,N); ((1+alpha)\*AyO+AyE+AyW+AyN+AyS); Inf\*ones(1,N)]; %AyO = (2:end,:), AyN = (1:end-1,:)

uh = [uh zeros(M,1)];

vh = [zeros(1,N); vh];

%Calculate link coefficients for all nodes

ApE = rho\*dy^2./sAx(:,2:end);

ApW = rho\*dy^2./sAx(:,1:end-1);

ApN = rho\*dx^2./sAy(1:end-1,:);

ApS = rho\*dx^2./sAy(2:end,:);

ApO = -(ApE+ApW+ApN+ApS);

Sp = rho\*((uh(:,2:end)-uh(:,1:end-1))\*dy+(vh(1:end-1,:)-vh(2:end,:))\*dx);

end

function [uh Resx] = Xsolver(alpha,AxO,AxE,AxW,AxN,AxS,Sx,u)

Rtol = 1E-20;

itrmax = 2;

[M N] = size(Sx);

%Reshape link coefficients

AxO = reshape(AxO,N\*M,1);

AxE = reshape(AxE,N\*M,1);

AxW = reshape(AxW,N\*M,1);

AxN = reshape(AxN,N\*M,1);

AxS = reshape(AxS,N\*M,1);

AxN = circshift(AxN,-1);

AxS = circshift(AxS,1);%need to check to make sure AN,AS are being shifted properly

AxE = circshift(AxE,M);

AxW = circshift(AxW,-M);

Ax = spdiags([AxW AxN AxO AxS AxE],[-M -1 0 1 M],N\*M,N\*M);

%Modify source for correction equation

Rx = reshape(Sx,M\*N,1)-Ax\*reshape(u(:,2:end),M\*N,1);

Sx = reshape(Rx,M,N);

Resx = norm(Rx);

Ax = Ax+diag(AxO\*alpha);

%Solve for u hat

[uh R] = ADIp(Ax,Sx,'-TDMA',Rtol,itrmax);

%append boundary condition

uh = [zeros(M,1) uh];

end

function [vh Resy] = Ysolver(alpha,AyO,AyE,AyW,AyN,AyS,Sy,v)

Rtol = 1E-20;

itrmax = 2;

[M N] = size(Sy);

%Reshape link coefficients

AyO = reshape(AyO,N\*M,1);

AyE = reshape(AyE,N\*M,1);

AyW = reshape(AyW,N\*M,1);

AyN = reshape(AyN,N\*M,1);

AyS = reshape(AyS,N\*M,1);

AyN = circshift(AyN,-1);

AyS = circshift(AyS,1);%need to check to make sure AN,AS are being shifted properly

AyE = circshift(AyE,M);

AyW = circshift(AyW,-M);

Ay = spdiags([AyW AyN AyO AyS AyE],[-M -1 0 1 M],N\*M,N\*M);

%Modify source for correction equation

Ry = reshape(Sy,M\*N,1)-Ay\*reshape(v(1:end-1,:),M\*N,1);

Sy = reshape(Ry,M,N);

Resy = norm(Ry);

Ay = Ay+diag(AyO\*alpha);

%Solve for v hat

[vh R] = ADIp(Ay,Sy,'-TDMA',Rtol,itrmax);

%append boundary condition

vh = [vh; zeros(1,N)];

end

function [pp Rp] = Psolver(ApO,ApE,ApW,ApN,ApS,Sp)

Rtol = 1E-20;

itrmax = 20;

[M N] = size(Sp);

%Reshape link coefficients

ApO = reshape(ApO,N\*M,1);

ApE = reshape(ApE,N\*M,1);

ApW = reshape(ApW,N\*M,1);

ApN = reshape(ApN,N\*M,1);

ApS = reshape(ApS,N\*M,1);

ApN = circshift(ApN,-1);

ApS = circshift(ApS,1);%need to check to make sure AN,AS are being shifted properly

ApE = circshift(ApE,M);

ApW = circshift(ApW,-M);

Ap = spdiags([ApW ApN ApO ApS ApE],[-M -1 0 1 M],N\*M,N\*M);

%Solve for u hat

[pp R] = ADIp(Ap,Sp,'-TDMA',Rtol,itrmax);

Rp = norm(R);

end

function [phi,R] = ADIp(X,S,method,Rtol,itrmax)

%Performs the Alternating Direction Implicit solver for a 2 dimensional

%system.

%Note: conversion from 2-D grid to 1-D array must be done as i,j -> k = M(i-1)+j,

%where there are M nodes in the y direction and N nodes in the x

%direction

%Code Version: 1.0 @ 2011-04-10

%Inputs:

% X: matrix for the x-derivative terms (M\*N,M\*N)

% S: source term, either matrix or scalar (M,N)

% method: '-TDMA' for tridiag systems, '-GaussSeidel' otherwise

% Rtol: Residual tolerance (optional; default 1E-5)

% itrmax: maximum number of iterations (optional; default 1E4)

%Outpus:

% phi: Solution to the problem

% R: L2norm of the Residuals at each iteration

if ~exist('Rtol','var')

Rtol = 1E-5;

end

if ~exist('itrmax','var')

itrmax = 1E4;

end

[M N] = size(S);

phi = zeros(M\*N,1);

S = reshape(S,M\*N,1);

R = zeros(1,itrmax);

R(1) = norm(X\*phi-S);

counter = 0;

AW = full([zeros(M,1); diag(X,-M)]);

AE = full([diag(X,M); zeros(M,1)]);

AN = full([diag(X,1); 0]);

AS = full([0; diag(X,-1)]);

AO = full(diag(X));

while (R(counter+1) >= Rtol) && (counter < itrmax)

%Row sweep

k = 1:M:1+M\*(N-1);

b = S(k)-AN(k).\*phi(k+1);

phi(k) = TDMAsolver(AW(k),AO(k),AE(k),b);

for j = 2:M-1

k = j:M:j+M\*(N-1); %determine nodal array points

b = S(k)-AN(k).\*phi(k+1)-AS(k).\*phi(k-1);

phi(k) = TDMAsolver(AW(k),AO(k),AE(k),b);

end

k = M:M:M+M\*(N-1); %determine nodal array points

b = S(k)-AS(k).\*phi(k-1);

phi(k) = TDMAsolver(AW(k),AO(k),AE(k),b);

%Column sweep

k = M\*(1-1)+1:M\*1;

b = S(k)-AE(k).\*phi(k+M);

phi(k) = TDMAsolver(AS(k),AO(k),AN(k),b);

for i = 2:N-1

k = M\*(i-1)+1:M\*i;

b = S(k)-AE(k).\*phi(k+M)-AW(k).\*phi(k-M);

phi(k) = TDMAsolver(AS(k),AO(k),AN(k),b);

end

k = M\*(N-1)+1:M\*N;

b = S(k)-AW(k).\*phi(k-M);

phi(k) = TDMAsolver(AS(k),AO(k),AN(k),b);

counter = counter + 1;

R(counter+1) = norm(X\*phi-S);

end

R = R(1:counter+1);

phi = reshape(phi,M,N);

end