Two Dimensional Coupled Flow Solver

Jacob Child

I, II: Representative Plots of the simulation

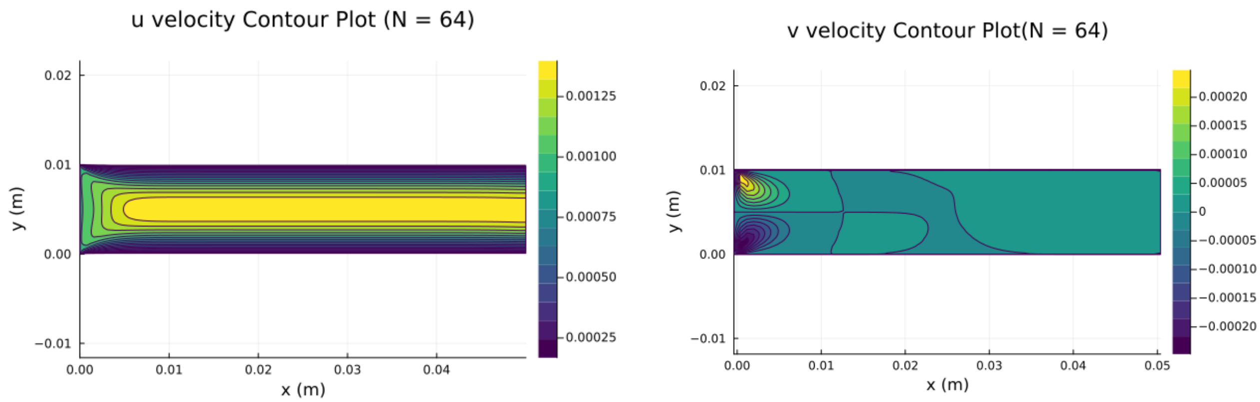


Figure 1: Contour plots of the x and y components of velocity (u and v respectively). Note that v is not quite symmetrical. Note: both color bars have units of m/s

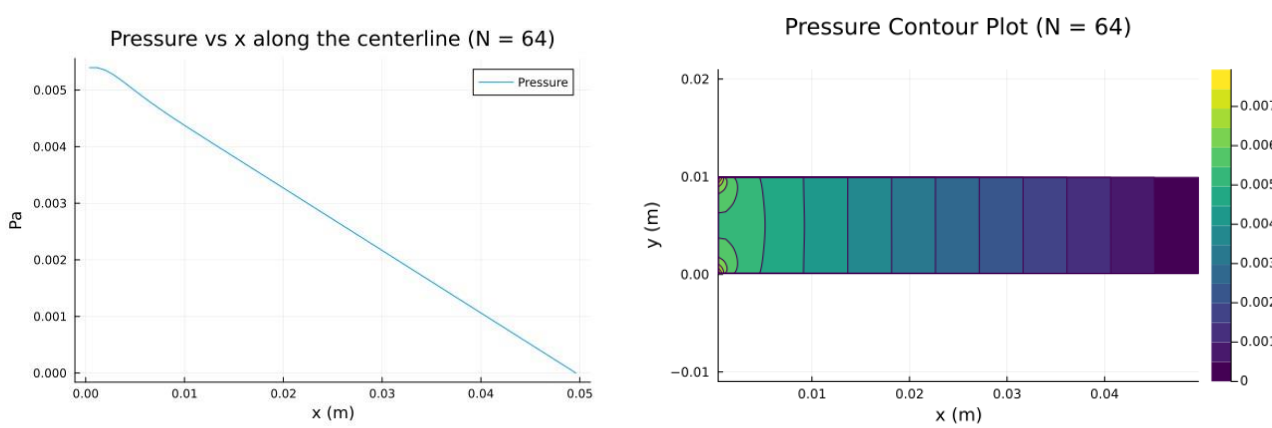


Figure 2: Pressure along the centerline (plot on the left) and within (plot on the right) the channel. It should be noted that there are an even number of nodes, so the two sets of nodes either side of the centerline were averaged to obtain this line. Note: the color bar in the right plot has units of Pa

Discussion: What was expected to be perfectly smooth and simple flow through a channel can have some complicated structures. The u-component of velocity (Figure 1 left) looks mostly how we would expect it to, similar to pipe flow. The region in the center of the channel is moving fastest as the boundary layers (shear stresses) are slowing down the edges, although the flow, and the boundary layers, appear to fully develop by about 0.0075m into the channel. The v-component of velocity (Figure 1 right) is especially insightful as it shows that there is a very slight component of velocity that is in the positive y direction throughout most of the flow. This likely comes due to the constriction caused at the entrance of the channel, adding vorticity to the flow and possibly some non-symmetry it appears. It is also possible that the non-symmetry comes from numerical or calculation error.

The pressure appears as we would expect as well (Figure 2), with the pressure gradually decreasing until it matches the exit pressure boundary condition. It can be seen in the contour plot that the same corners that cause y velocity and possibly some vorticity in the v velocity plot, also cause a large pressure spike as well due to the sudden constriction. The

<u>III. Grid Independence and Convergence</u>

To verify grid independence, the number of nodes of the grid was increased and the results tabulated. The results both visually and numerically begin to show grid convergence. The final converged value was N = 64, which leads to a mesh of 4096 pressure points. Centerline pressure was plotted and compared as seen in Figure 3. The center line pressure does not yet seem to converge all the way and is still decreasing, however is quite close. When looking at the u-velocity profile at an x location of 0.025 (the midline), you can begin to see flow structure convergence that looks very similar to pipe flow. The v-velocity profile 0.15cm above the bottom wall was also plotted and can be seen in the appendix. To numerically compare the convergence of pressure, u, and v velocity between grid sizes, the maximums of each were compared. Table 1 shows the percent the maximum centerline pressure changes with increased discretization, as well as the same for u and v velocities. It should be noted that u velocity can be considered fully converged, Pressure is quite close, and v velocity is also quite close to converging, although it is the furthest off of the bunch. This convergence was considered satisfactory for the mostly qualitative comparison that will be made with commercial CFD codes.

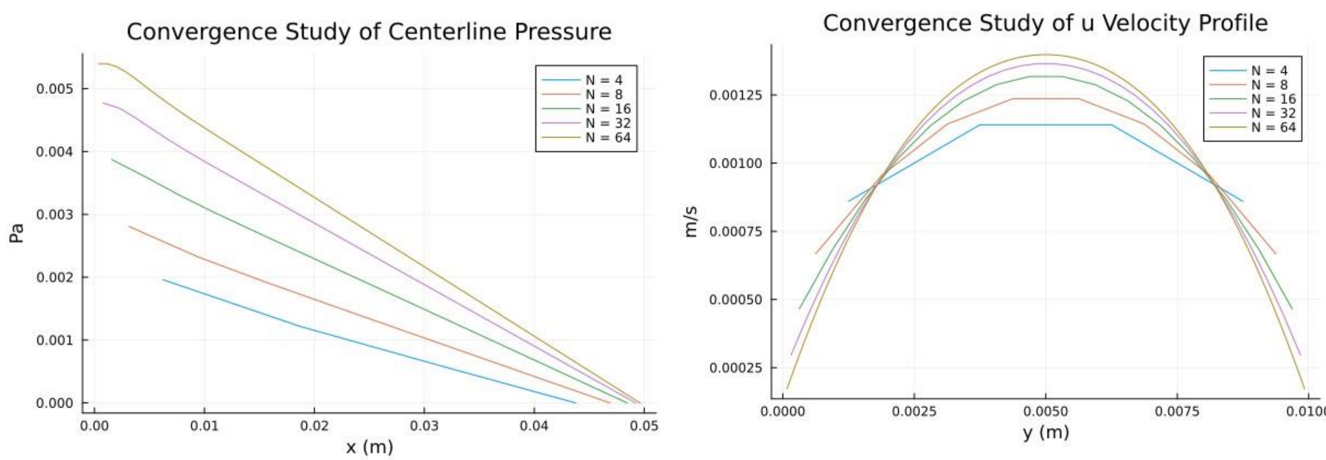


Figure 3: Chosen grid convergence study metrics. Centerline pressure profile and v velocity profile.

Table 1: Grid convergence table showing the percent difference between values when increasing the number of nodes.

Pressure Grid width N (in N x N mesh)	**4**	**8**	**16**	**32**	**64**
Pressure Percent Change	NA	30.27%	27.46%	18.86%	11.54%
U Velocity Percent Change	NA	7.75%	6.15%	3.46%	2.35%
V Velocity Percent Change	NA	98.92%	51.71%	30.85%	16.47%

In order to ensure convergence of the inner loop solving for u, v, and pressure, this iterative portion of the code was put within a while loop. To ensure that x momentum, y momentum, and pressure had all converged, the difference between the maximum values of each between iterations had to be less than 1e-6 before it was considered converged. This worked quite well and yielded better looking results when compared to a lower tolerance.

IV. Summary of Numerical Parameters

Table 2: Summary of parameters used in the coded 2D coupled flow solver calculations

Number of Pressure Cells	**u-velocity under-relaxation factor**	**v-velocity under-relaxation factor**	**Pressure under-relaxation factor**	**Tolerance**
4096	0.5 (until after 200 iterations -> 0.499)	0.5 (until after 200 iterations -> 0.499)	1.0 (until after 200 iterations -> 0.999)	1e-6

The parameters shown in Table 2 were chosen because they led to momentum and pressure plots that looked correct after converging under tolerance. The relaxation factors were originally changed more aggressively, the most aggressive was subtracting about 0.01-0.025 from each relaxation factor every 20 iterations. However, it was found that over aggressively reducing the relaxation factors led to improper results, so the bulk of the iterations were left to normal relaxation factor values. The reason for the number of pressure cells and tolerance were discussed above.

V. Validation with Commercial Code

A. Verification of Commercial Code

Star CCM + was the commercial code chosen to validate the 2D coupled flow solver. The Reynolds number of this flow is 50, and the following solver and physics models were chosen to correctly represent the flow: Steady, Laminar, liquid, constant density, segregated flow. Twenty prism layers to a height of 0.025cm off of each wall were used to properly capture the boundary layer growth. The same physics conditions and initial conditions as the 2D coupled flow solver were used. In order to verify that this commercial code was performing correctly a mesh convergence study was done using the same parameters as the 2D coupled flow solver: centerline entrance pressure and u velocity at the center midline (convergence Figures can be seen in the Appendix). Each simulation was also run until the residuals decreased and flattened out. Similar to the coded 2D coupled flow solver, it was found that the v velocity profile, whether at the centerline, or 0.15cm above the lower wall, was not a good convergence measure due to the variance at the entrance and exit of the channel that discretization causes (further discussion in the Appendix). The centerline entrance pressure and u velocity at the center midpoint are compared numerically and show good convergence. The percent that the pressure and u velocity values changed in between changes of the base mesh size can be seen in Table 2. All of the changes are within less than a percent, which shows that the mesh (as seen in Figure 4) is fine enough, and the output values can be used to compare with the coded 2D coupled flow solver. The base mesh size of 0.01m was used for the comparisons.

Table 3: Star CCM+ Mesh convergence study results. The percent the result changed with the decreased mesh base size.

Base Size:	.08 (m)	.04 (m)	.02 (m)	.01 (m)
Pressure Percent Change	NA	-0.32%	-0.57%	0.07%
U Velocity Percent Change	NA	-0.26%	-0.33%	-0.07%

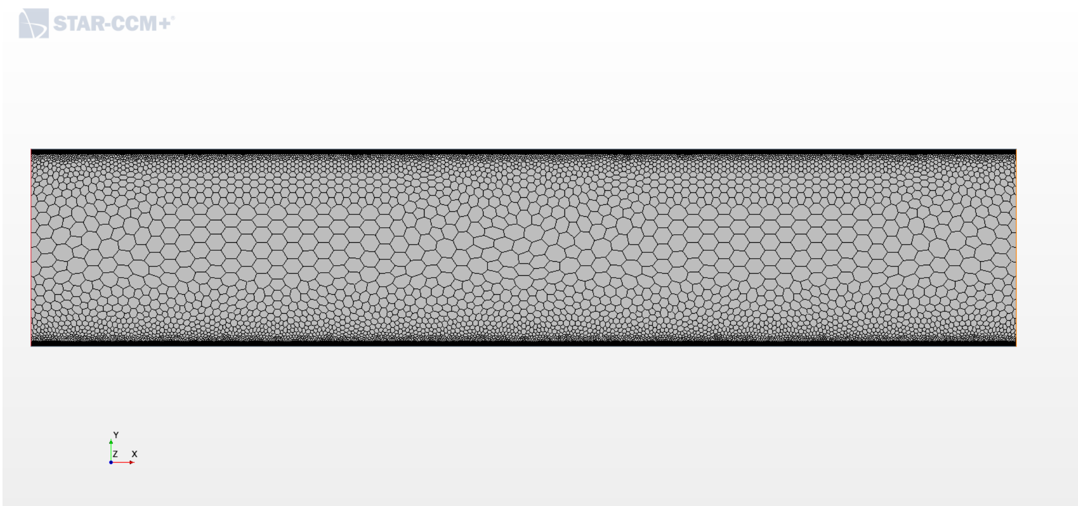


Figure 4: The converged mesh in Star CCM + with a base size of 0.01m

B. Result Comparison and Validation

The coded two dimensional coupled flow solver matched quite well with the Star CCM+ results. Figure 5 shows the qualitative top (Star CCM +) and bottom (coded 2D solver) comparison of the u-velocity contour plots. The general u-velocity flow trends match quite well. It can be seen that the flow develops similar profiles at about similar lengths down the channel, and the width of the inner high speed flow regions are similar. More qualitative result comparisons can be seen in the appendix. Figure 6 is what validates the u – velocity profile found by the coded 2D flow solver. This figure shows the u-velocity profile found at the midline of the solver and Star CCM + and overlays them. In general the agreement is quite good. The step-like behavior of the Star CCM + (CFD) line comes from the larger mesh cell sizes found in the middle of the channel on the CFD mesh. The profile the CFD shows is slightly more peaked and narrower, showing a more fully developed flow, with the center flowing even faster, and the sides even slower (the presence of prism layers definitely helps there). It is likely that with more cells, and similar prism layers, the coded 2D flow solver would match even better.

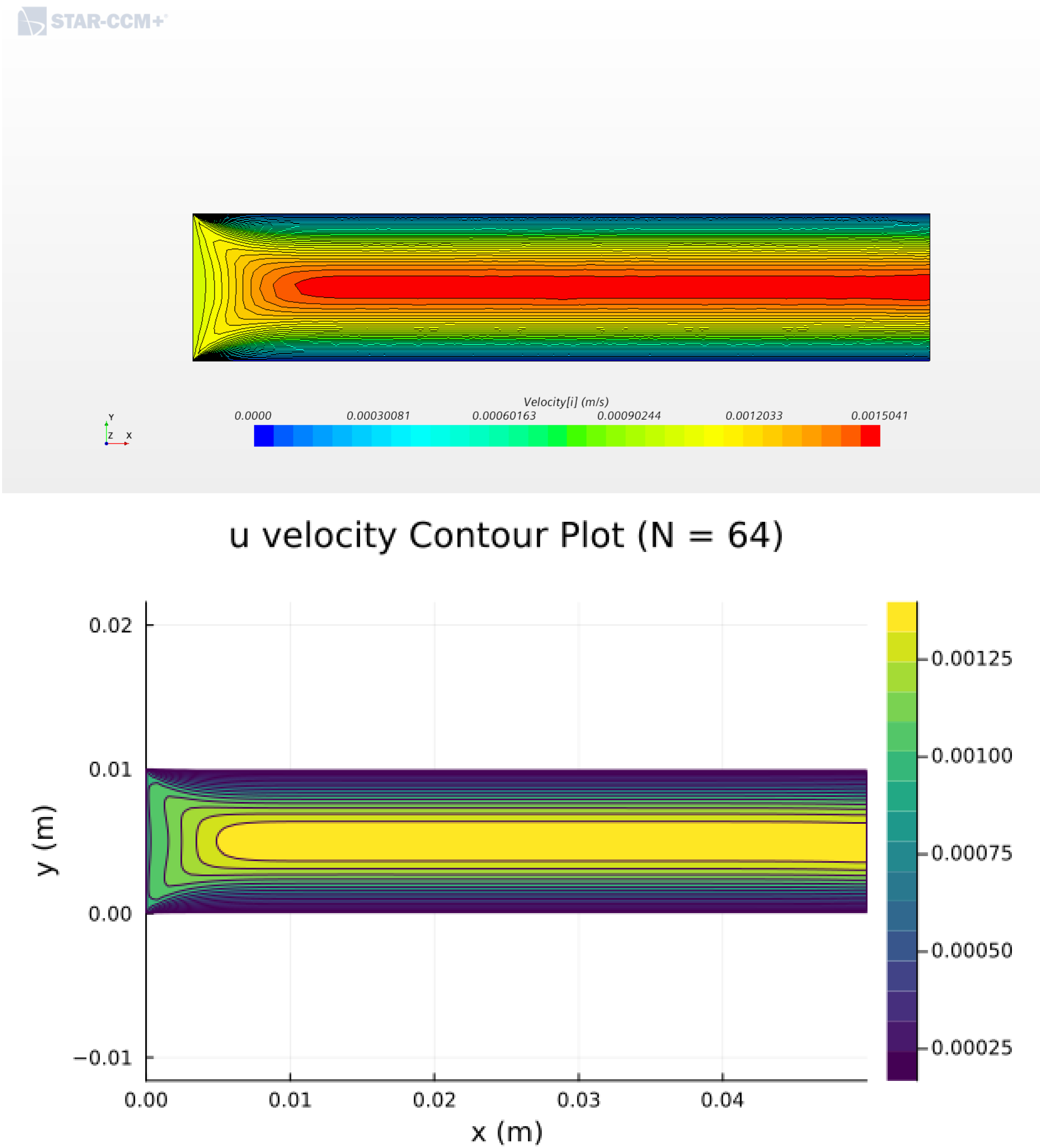


Figure 5: Contour plots of u - velocity from Star CCM + (top) and the coded 2D flow solver (bottom). Note: the color bar on the bottom plot has units of m/s

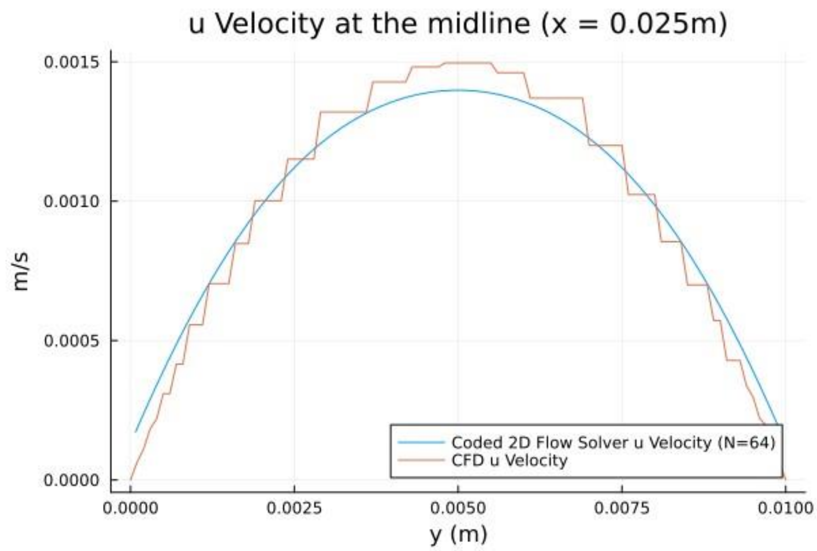


Figure 6: Comparison of the u-velocity profiles at the midline (x = 0.025m) of the channel.

To further validate the coded 2D flow solver the v – velocity profiles 0.0015m above the bottom wall were also plotted and compared as seen in Figure 7. The profiles are quite similar, and yet again it seems the coded 2D flow solver can be considered valid. There are a few small differences between the two profiles however. The v-velocity just after the entrance corner spikes much higher in Star CCM+ than in the coded solver. This is yet again likely due to the finer discretization of the commercial code. The general flow shape and behavior is capture quite well by the coded solver, including the slight increase and then decrease in v-velocity near the exit, however the coded solver seems to start earlier, peak higher, and not dip as low. This could come because it is still not quite converged (see part 1) and could still be discretized further.

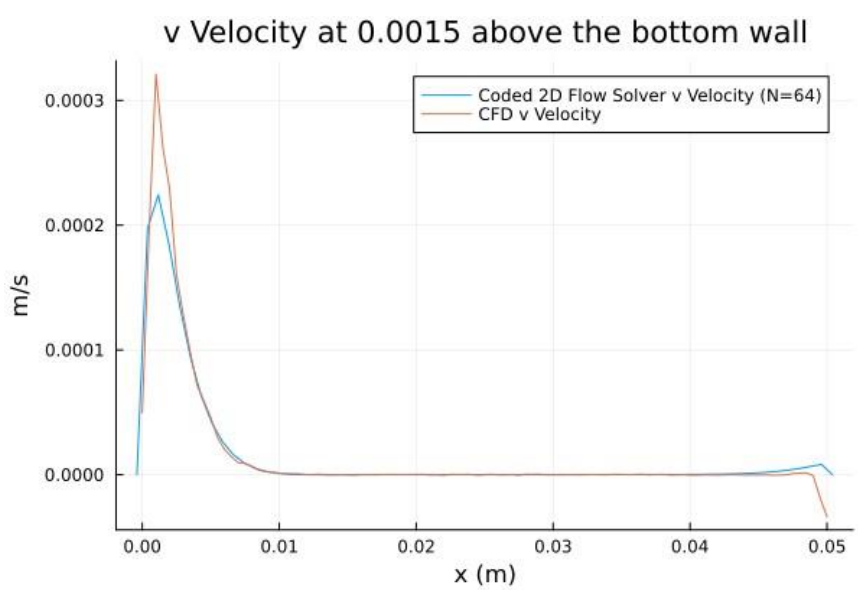


Figure 7: Comparison of v-velocity profiles 0.0015m above the bottom wall between Star CCM + (CFD) and the coded solver

The last portion left to validate is the centerline pressure. Figure 8 shows the comparison of the centerline pressures found with the coded solver and the commercial code. The shape of the agreement is quite good, however the coded solver in general under predicts the pressure throughout the channel. The initial difference at the entrance is less than 16% however and gets better further along the channel. The lack of smoothness in the commercial code line (CFD Pressure) comes from the discretization through the center of the Star CCM + mesh being more coarse.

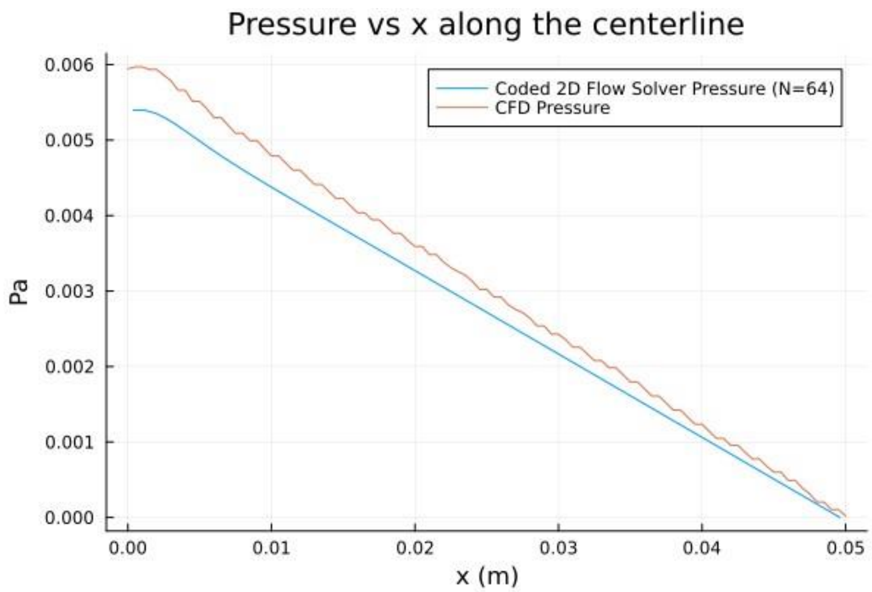


Figure 8: Comparison between centerline pressure values of Star CCM+ (CFD Pressure) and the coded 2D flow solver.

Conclusion

In conclusion the coded 2D coupled flow solver for the channel performed quite well and is verified through u and v velocity component convergence as well as pressure convergence. The results are further validated by the Star CCM + commercial CFD code that was also verified through convergence in u velocity and pressure. The results compare quite well and further similarity could likely happen with increased discretization of the coded solver. Improvements could be made in the solver's calculation speed (the code can be further optimized) and discretization methods (non-grid like spacing would be useful). The adaptive relaxation factor was shown to not perform consistently between grid sizes and leaving the relaxation factors at their initial values and allowing the solver to iterate longer yielded better results. The u-velocity profiles and behavior, as well as the pressure behavior was readily understood when compared to channel or pipe flow and as a result of developing flow and boundary layer growth. The v-velocity profiles were more difficult to understand and semi-coupled to the grid/mesh size. Due to the flow being funneled into the channel, the corners cause some v components of velocity that then get propagated through the rest of the channel flow. Overall it was a fun project!

APPENDIX

Coded solver grid convergence study: note that the spike grows with discretization close to the entrance. The point at which the flow appears to go back to 0 v-velocity seems to have converged just before 0.01m and there is little change between discretization, however this is a difficult convergence metric to use, and combined with the spike makes it even more difficult, thus it is still changing.

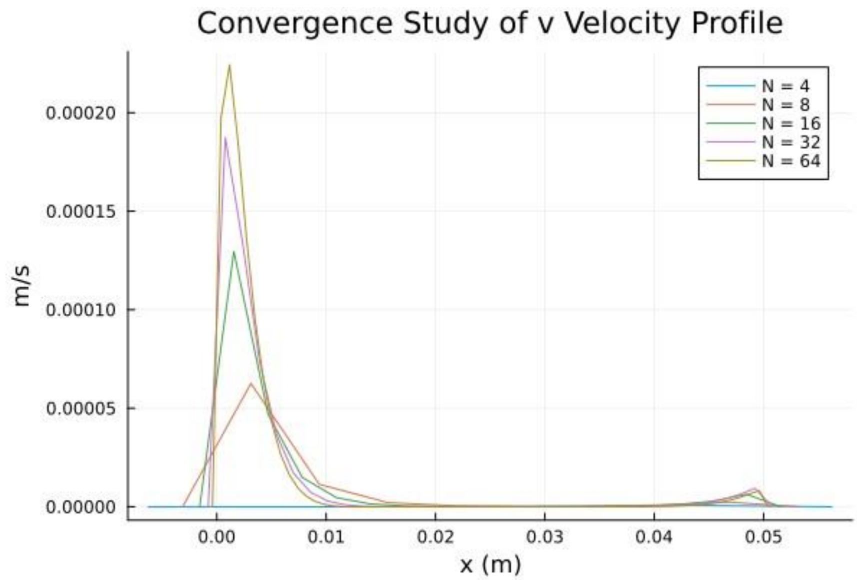
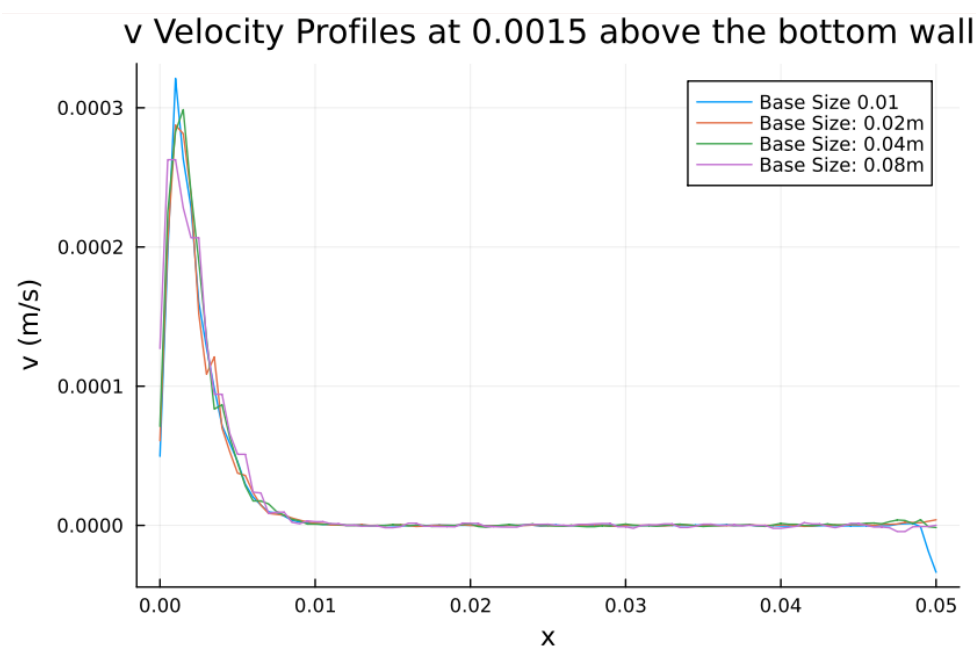


Figure 9: Part of the grid convergence study for the coded solver

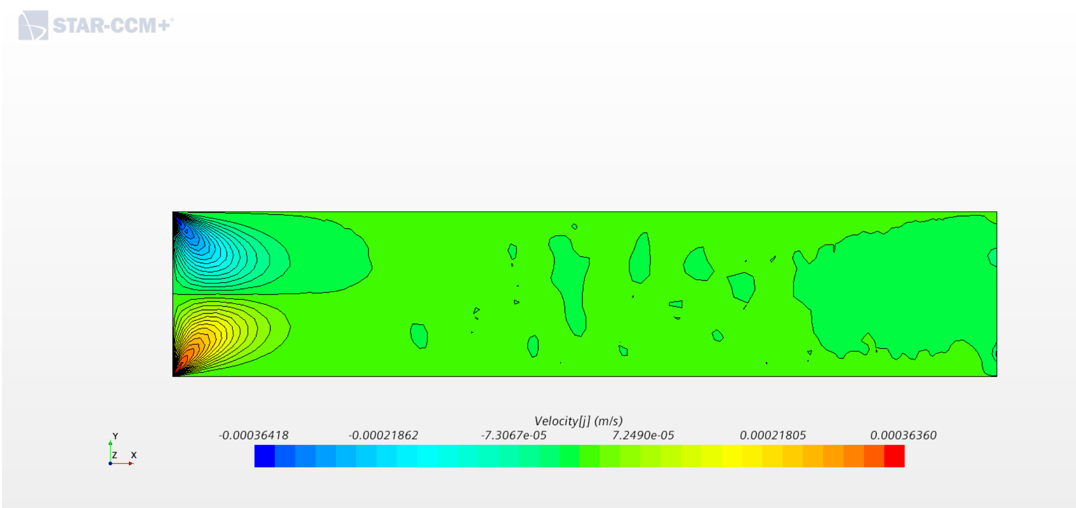
Star CCM+ convergence plots

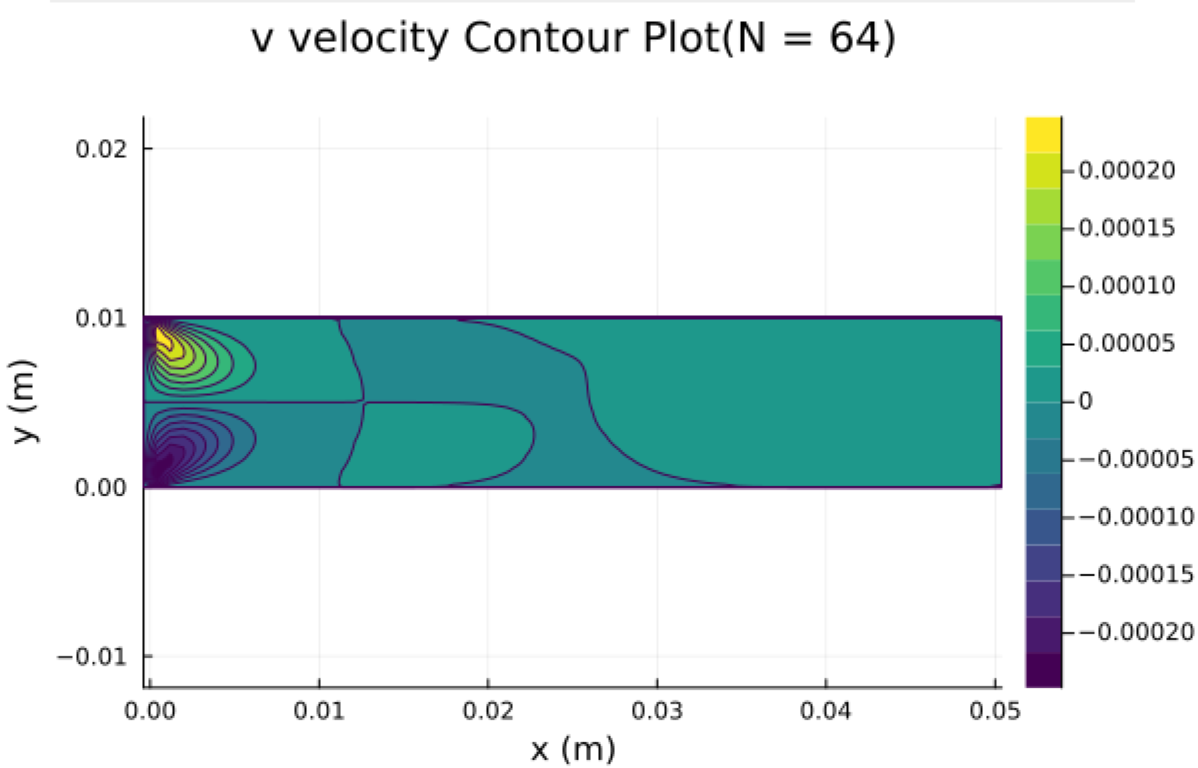


The centerline pressures at each grid size and the u velocity profiles at each grid size were not exported, just the relevant values to create the convergence table seen in the above relevant section.

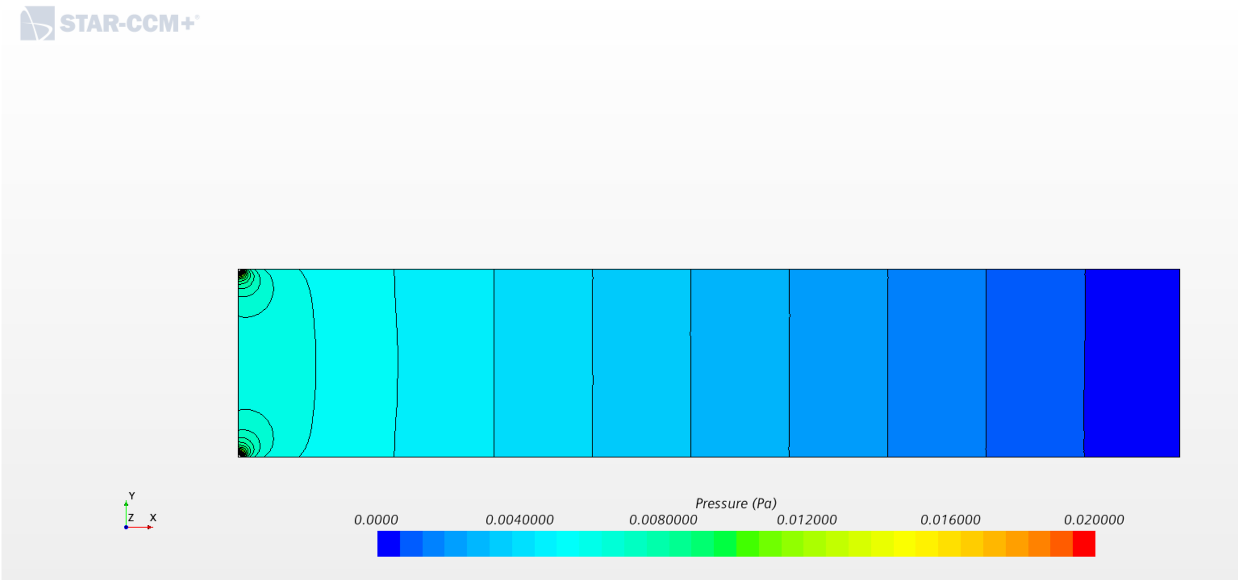
Further Qualitative Comparison

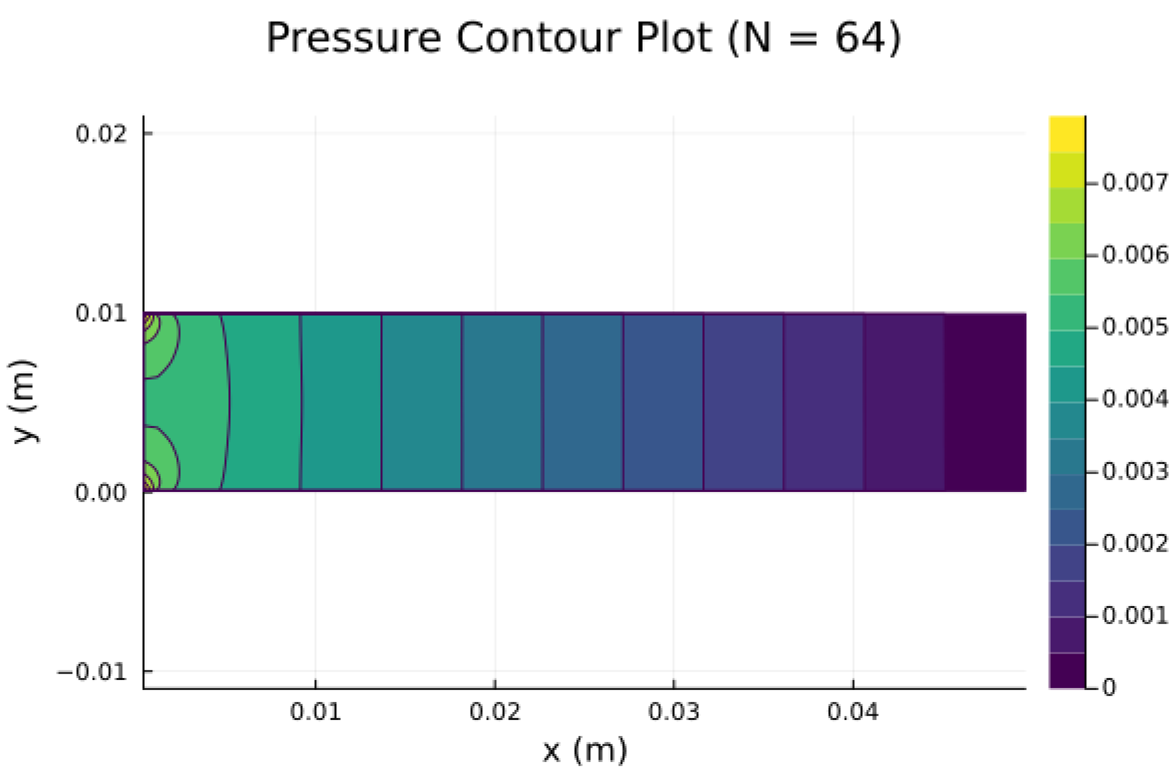
v-velocity comparison: the corners seem to match, my grid misses the small differences in the middle





Pressure Comparison: These appear to match up quite well, although the centerline pressure plot in the main section would better show that the coded solver slightly under predicts pressure.





Code:

Note: the first file/set of code is the upper level structure, that calls all of the functions contained in the second set. The comparison code for all of the plots etc is in a mix of places, at the end of the first file, and in third file, as well as some commands that probably weren't recorded.

FinalProjectTwoDCoupledFlowSolver\TwoDSimpleFuncs.jl

```julia
#=
TwoDSimpleFuncs.jl
Jacob Child
April 11th, 2024
Psuedocode: functions needed to assist in the SIMPLE CFD algorithm, also includes functions specific to the final 2D channel flow problem
=#

"""
uABMaker #TODO: finish the header
"""
function uABMaker(uf, vf, Pf; dxf = dx, dyf = dy, μf = μ, ρf = ρ, uinf = Vin, αuf = αu, wallflag = "internal")
    #uf is a 3 row matrix
    #vf is a 2 row matrix
    #Pf is a Vector (1 row matrix), it should be the same row as the middle row in the uf matrix
    #!this code is made for a contant dxf (dx) and dyf (dy), but could be easily adjusted to be adaptable
    #East and West control surface values needed for the *internal velocity nodes* only
    De = μf/dxf
    Dw = μf/dxf
    Dn = μf/dyf
    Ds = μf/dyf

    #Initialize Ae, Aw, Ap, b, and d vectors to be put in the matrix later, use i,j I,J notation
    ai1J = zeros(size(uf,2)) #Ae
    aim1J = zeros(size(uf,2)) #Aw, aim1 = a(i-1)
    aiJ1 = zeros(size(uf,2)) #An
    aiJm1 = zeros(size(uf,2)) #As
    aiJ = zeros(size(uf,2)) #Ap
    BiJ = zeros(size(uf,2)) #b

    #Internal node for loop
    J = 2 #ie the primary row we are looking at is the 2nd row of the 3 pulled in for u
    internalend = length(aiJ)-1
    for i in 2:internalend #this leaves the boundaries out
        Fe = ρf / 2 * (uf[J,i+1] + uf[J, i])
        Fw = ρf / 2 * (uf[J,i] + uf[J, i-1])
        Fn = ρf / 2 * (vf[1,i+1] + vf[1,i]) #this row callout appears backwards from the equation. That is because I am starting in the top left, so j+1 is row 1 of v and j is row 2 of v
        Fs = ρf / 2 * (vf[2,i+1] + vf[2,i])

        ai1J[i] = (De*dyf + max(-Fe, 0)*dyf) #* αuf
        aim1J[i] = (Dw*dyf + max(Fw, 0)*dyf) #* αuf
        aiJ1[i] = (Dn*dxf + max(-Fn, 0)*dxf) #* αuf
        aiJm1[i] = (Ds*dxf + max(Fs, 0)*dxf) #* αuf
        aiJ[i] = (ai1J[i] + aim1J[i] + aiJ1[i] + aiJm1[i] +(Fe - Fw)*dyf + (Fn - Fs)*dxf) / αuf
        if wallflag == "lower"
            aiJm1[i] = 0
            aiJ[i] = (ai1J[i] + aim1J[i] + aiJ1[i] + aiJm1[i] +(Fe - Fw)*dyf + (Fn - Fs)*dxf + μf * dxf / (dyf/2)) / αuf
```

```julia

        elseif wallflag == "upper"
            aiJ1[i] = 0
            aiJ[i] = (ai1J[i] + aim1J[i] + aiJ1[i] + aiJm1[i] +(Fe - Fw)*dyf + (Fn - Fs)*dxf + μf * dxf / (dyf/2) ) / αuf
        end

        BiJ[i] = ((Pf[i-1] - Pf[i])*dyf + 0 + (1-αuf)/αuf * aiJ[i] * uf[J,i] * αuf)  #this is the u momentum equation with the underrelaxation factor added in

    end

    #println("Ap: ", Ap)
    #Boundary conditions/external nodes
    #println(ai1J)
    #node 1

    ai1J[1] = 0
    aim1J[1] = 0
    aiJ1[1] = 0
    aiJm1[1] = 0
    aiJ[1] = 1
    BiJ[1] = uinf

    #node 5
    ai1J[end] = 0
    aim1J[end] = 1
    aiJ1[end] = 0
    aiJm1[end] = 0
    aiJ[end] = 1
    BiJ[end] = 0


    return ai1J, aim1J, aiJ1, aiJm1, aiJ, BiJ
end



function vABMaker(uf, vf, Pf; dxf = dx, dyf = dy, μf = μ, ρf = ρ, uinf = Vin, vinletf = 0, αvf = αv , wallflag = "internal")
    #uf is a 2 row matrix
    #vf is a 3 row matrix amd row 2 is the row of interest I believe
    #Pf is a 2 row matrix, same location as uf
    #!this code is made for a contant dxf (dx) and dyf (dy), but could be easily adjusted to be adaptable
    #East and West control surface values needed for the *internal velocity nodes* only
    De = μf/dxf
    Dw = μf/dxf
    Dn = μf/dyf
    Ds = μf/dyf
    #Initialize Ae, Aw, Ap, b, and d vectors to be put in the matrix later, use i,j I,J notation
    ai1J = zeros(size(vf,2)) #Ae
    aim1J = zeros(size(vf,2)) #Aw, aim1 = a(i-1)
    aiJ1 = zeros(size(vf,2)) #An
    aiJm1 = zeros(size(vf,2)) #As
    aiJ = zeros(size(vf,2)) #Ap
```

```
    BiJ = zeros(size(vf,2)) #b

    #Internal node for loop
    internalend = length(aiJ)-1

    for i in 2:internalend #this leaves the boundaries out
        Fe = ρf / 2 * (uf[1,i] + uf[2, i])
        Fw = ρf / 2 * (uf[1,i-1] + uf[2, i-1])
        Fn = ρf / 2 * (vf[2,i] + vf[1,i]) #this row callout appears backwards, but should be correct
        Fs = ρf / 2 * (vf[3,i] + vf[2,i])

        if wallflag == "lower" || wallflag == "upper"
            ai1J[i] = 0
            aim1J[i] = 0
            aiJ1[i] = 0
            aiJm1[i] = 0
            aiJ[i] = 1
            BiJ[i] = 0
        else
        ai1J[i] = (De*dyf + max(-Fe, 0)*dyf) #* αvf
        aim1J[i] = (Dw*dyf + max(Fw, 0)*dyf) #* αvf
        aiJ1[i] = (Dn*dxf + max(-Fn, 0)*dxf) #* αvf
        aiJm1[i] = (Ds*dxf + max(Fs, 0)*dxf) #* αvf
        aiJ[i] = (ai1J[i] + aim1J[i] + aiJ1[i] + aiJm1[i] +(Fe - Fw)*dyf + (Fn - Fs)*dxf) / αvf

        BiJ[i] = (Pf[2,i-1] - Pf[1,i-1])*dxf  + 0 + (1-αvf)/αvf * aiJ[i] * vf[2,i] * αvf
        end
    end

    #println("Ap: ", Ap)
    #Boundary conditions/external nodes
    #node 1
    ai1J[1] = 0
    aim1J[1] = 0
    aiJ1[1] = 0
    aiJm1[1] = 0
    aiJ[1] = 1
    BiJ[1] = vinletf #no y component of inlet velocity

    #node 6
    ai1J[end] = 0
    aim1J[end] = 1
    aiJ1[end] = 0
    aiJm1[end] = 0
    aiJ[end] = 1
    BiJ[end] = 0


    return ai1J, aim1J, aiJ1, aiJm1, aiJ, BiJ
end


function pPrimeABMaker(uapf, vapf, usnewf, vsnewf, Pf; dxf = dx, dyf = dy, αuf = αu, αvf = αv, αpf = αp, ρf = ρ, wallflag = "internal")
    #uapf is a single row vector, at the same row as P
```

```
    #vapf is a 2 row matrix, the row above the and below the P row of interest
    #usnewf is a single row vector at the same row as P
    #vsnewf is a 2 row matrix, the row above and below the P row of interest
    #!this code is made for a contant dxf (dx) and dyf (dy), but could be easily adjusted to be adaptable
    #we are solving at a single row of P

    #Initialize Ae, Aw, Ap, b, and d vectors to be put in the matrix later, use i,j I,J notation

    aIp1J = zeros(size(Pf,1)) #Ae
    aIm1J = zeros(size(Pf,1)) #Aw, aim1 = a(i-1)
    aIJp1 = zeros(size(Pf,1)) #An
    aIJm1 = zeros(size(Pf,1)) #As
    aIJ = zeros(size(Pf,1)) #Ap
    bpIJ = zeros(size(Pf,1)) #b

    #Internal node for loop
    internalend = size(aIJ,1)-1
    for i in 1:internalend #this leaves the boundaries out
        dip1J = dyf / uapf[i+1] #not multiplied by αuf as uapf already has it
        diJ = dyf / uapf[i]
        dIjp1 = dxf / vapf[1,i+1] #not multiplied by αvf as vapf already has it
        dIj = dxf / vapf[2,i+1]

        if wallflag == "upper" && uapf[i] == 1
            diJ = dyf / uapf[i] * αuf
            dIjp1 = dxf / vapf[1,i+1] * αvf
        end

        if wallflag == "upper" && uapf[i] != 1
            dIjp1 = dxf / vapf[1,i+1] * αvf
        end

        if wallflag == "internal" && uapf[i] == 1
            diJ = dyf / uapf[i] * αuf
        end

        if wallflag == "lower" && uapf[i] == 1
            diJ = dyf / uapf[i] * αuf
            dIj = dxf / vapf[2,i+1] * αvf
        end

        if wallflag == "lower" && uapf[i] != 1
            dIj = dxf / vapf[2,i+1] * αvf
        end

        aIp1J[i] = dip1J * ρf * dyf
        aIm1J[i] = diJ * ρf * dyf
        aIJp1[i] = dIjp1 * ρf * dxf
        aIJm1[i] = dIj * ρf * dxf

        # #wall corrections
        # if wallflag == "lower"
        #     aIJm1[i] = 0 #these values are still included in aIJ, but are not used in the calculation overall for P
        # elseif wallflag == "upper"
```

```
        #     aIJp1[i] = 0
        # end

        aIJ[i] = aIp1J[i] + aIm1J[i] + aIJp1[i] + aIJm1[i]

        bpIJ[i] = -(ρf*usnewf[i+1]*dyf) + (ρf*usnewf[i]*dyf) - (ρf*vsnewf[1,i+1]*dxf) + (ρf*vsnewf[2,i+1]*dxf)


        #wall corrections
        if wallflag == "lower"
            aIJm1[i] = 0 #these values are still included in aIJ, but are not used in the calculation overall for P
        elseif wallflag == "upper"
            aIJp1[i] = 0
        end
    end

    #Boundary conditions/external nodes
    #node 5
    aIp1J[end] = 0
    aIm1J[end] = 0
    aIp1J[end-1] = 0
    aIJp1[end] = 0
    aIJm1[end] = 0
    aIJ[end] = 1
    bpIJ[end] = 0 #this is the exit condition I believe, pressure outlet

    #corner corrections
    if wallflag == "lower"
        aIm1J[1] = 0 #first node by the inlet has no aw
        aIp1J[end-1] = 0 #next to last node by the outlet has no ae
    elseif wallflag == "upper"
        aIm1J[1] = 0 #first node by the inlet has no aw
        aIp1J[end-1] = 0 #next to last node by the outlet has no ae
    end

    return aIp1J, aIm1J, aIJp1, aIJm1, aIJ, bpIJ

end


function iterator(u,v,P;)
    #Begin to solve, to start I will do 1 iteration at a time
    #preallocate us: east, west, north, south, B
    aues, auws, auns, auss, aups, Bus = [zero(u) for _ in 1:6]
    #preallocate vs: east, west, north, south, B
    aves, avws, avns, avss, avps, Bvs = [zero(v) for _ in 1:6]
    #Preallocate Ps: east, west, north, south, B
    apes, apws, apns, apss, apps, Bps = [zero(P) for _ in 1:6]

    #upper wall
    aues[1,:], auws[1,:], auns[1,:], auss[1,:], aups[1,:], Bus[1,:] = uABMaker(u[1:3,:], v[1:2,:], P[1,:], wallflag = "upper")

    #internal  for loop only
    for i in axes(u[1:end-2,:],1)
```

```
    aues[i+1,:], auws[i+1,:], auns[i+1,:], auss[i+1,:], aups[i+1,:], Bus[i+1,:]  = uABMaker(u[i:i+2,:], v[i+1:i+2,:], P[i+1,:])
    end

    #bottom wall
    aues[end,:], auws[end,:], auns[end,:], auss[end,:], aups[end,:], Bus[end,:] = uABMaker(u[end-2:end,:], v[end-1:end,:], P[end,:], wallflag = "lower")

    Au = AMaker2D(reverse(aups,dims=1),reverse(aues,dims=1),reverse(auws,dims=1), reverse(auns,dims=1),reverse(auss,dims=1),size(u))
    u1 = reverse(reshape(Au \ vec(reverse(Bus,dims=1)'),reverse(size(u)))',dims=1)
    #mdot correction
    mdotinapparent = sum(u1[:,1])
    mdotoutappar = sum(u1[:,end])
    u1[:,end] = u1[:,end] * mdotinapparent / mdotoutappar

    #v calculations
    #upper wall
    aves[1,:], avws[1,:], avns[1,:], avss[1,:], avps[1,:], Bvs[1,:] = vABMaker(u1[1:2,:], v[1:3,:], P[1:2,:], wallflag = "upper")
    #internal  for loop only
    for i in axes(v[1:end-2,:],1)
        aves[i+1,:], avws[i+1,:], avns[i+1,:], avss[i+1,:], avps[i+1,:], Bvs[i+1,:] = vABMaker(u1[i:i+1,:], v[i:i+2,:], P[i:i+1,:])

    end

    #bottom wall
    aves[end,:], avws[end,:], avns[end,:], avss[end,:], avps[end,:], Bvs[end,:] = vABMaker(u1[end-1:end,:], v[end-2:end,:], P[end-1:end,:], wallflag = "lower")

    Av = AMaker2D(reverse(avps,dims=1),reverse(aves,dims=1),reverse(avws,dims=1), reverse(avns,dims=1),reverse(avss,dims=1),size(v))
    v1 = reverse(reshape(Av \ vec(reverse(Bvs,dims=1)'),reverse(size(v)))', dims=1)

    #P calculations
    #upper wall
    apes[1,:], apws[1,:], apns[1,:], apss[1,:], apps[1,:], Bps[1,:] = pPrimeABMaker(aups[1,:] ,avps[1:2,:],u1[1,:],v1[1:2,:],P[1,:], wallflag = "upper")
    #internal  for loop only
    for i in axes(P[1:end-2,:],1)
    apes[i+1,:], apws[i+1,:], apns[i+1,:], apss[i+1,:], apps[i+1,:], Bps[i+1,:] = pPrimeABMaker(aups[i+1,:],avps[i+1:i+2,:],u1[i+1,:],v1[i+1:i+2,:],P[i+1,:])
    end
    #bottom wall
    apes[end,:], apws[end,:], apns[end,:], apss[end,:], apps[end,:], Bps[end,:] = pPrimeABMaker(aups[end,:],avps[end-1:end,:],u1[end,:],v1[end-1:end,:],P[end,:], wallflag = "lower")


    AP = AMaker2D(reverse(apps,dims=1),reverse(apes,dims=1),reverse(apws,dims=1), reverse(apns,dims=1),reverse(apss,dims=1),size(P))
    P1 = reverse(reshape(AP \ vec(reverse(Bps,dims=1)'),reverse(size(P)))', dims=1)

    #Calculate the new P, u, v
    #Pnew = P .+ αp .* P1
    Pnew = deepcopy(P1)
    Pnew[:,1:end-1] = P[:,1:end-1] .+ αp .* P1[:,1:end-1]
    #println("Pnew: ", Pnew)
```

```julia
    #throw("error")
    diJ = dy ./ aups
    dIj = dx ./ avps
    unew = deepcopy(u1)
    vnew = deepcopy(v1)

    unew[:,2:end-1] = u1[:,2:end-1] .+ diJ[:,2:end-1] .* -diff(P1,dims=2)
    vnew[2:end-1,2:end-1] = v1[2:end-1,2:end-1] .+ dIj[2:end-1,2:end-1] .* diff(P1,dims=1)
    #enforce the boundary condition
    vnew[:,end] .= 0


    return unew, vnew, Pnew, Au, Av, AP
end



function Converger(ustartf, vstartf, Pstartf;)
    #make all the keyword arguments global in this scope

    #initialize variables to store
    ustore, vstore, Pstore = [], [], []
    Aus, Avs, Aps = [], [], []
    Bus, Bvs, Bps = [], [], []

    ustore = push!(ustore, ustartf)
    vstore = push!(vstore, vstartf)
    Pstore = push!(Pstore, Pstartf)
    u1, v1, P1, Au1, Av1, Ap1 = iterator(ustore[end], vstore[end], Pstore[end])
    push!(ustore, u1)
    push!(vstore, v1)
    push!(Pstore, P1)

    push!(Aus, Au1)
    push!(Avs, Av1)
    push!(Aps, Ap1)

    #iterate
    #while !isapprox(1+maximum(Aus[end] * ustore[end-1] - Bus[end]), 1, atol = 1e-5) ||
!isapprox(1+maximum(Avs[end] * vstore[end-1] - Bvs[end]), 1, atol = 1e-5) ||
!isapprox(1+maximum(Aps[end] * Pstore[end-1] - Bps[end]), 1, atol = 1e-5)
    tolerance = 1e-6

    counter = 0
    gcounter = 0

    while maximum(ustore[end] - ustore[end-1]) > tolerance || maximum(vstore[end] -
vstore[end-1]) > tolerance || maximum(Pstore[end] - Pstore[end-1]) > tolerance
    #while max(abs(Aus[end]*ustore[end] - Bus[end])) > 1e-5 || max(abs(Avs[end]*vstore[end] -
Bvs[end])) > 1e-5
        counter += 1
        gcounter += 1
        u2, v2, P2, Au2, Av2, Ap2 = iterator(ustore[end], vstore[end], Pstore[end])
        push!(ustore, u2)
        push!(vstore, v2)
        push!(Pstore, P2)
        push!(Aus, Au2)
```

```
        push!(Avs, Av2)
        push!(Aps, Ap2)
        #to keep As and bs from growing huge
        popfirst!(Aus)
        popfirst!(Avs)
        popfirst!(Aps)
        # #!for big run only
        # popfirst!(ustore)
        # popfirst!(vstore)
        # popfirst!(Pstore)
        # #!for big run only

        #actively update the relaxation Factors
        val = .001 #works at N=64 and .025
        if counter > 200 && αu > val
            global αu, αv, αp
            αu = αu - val
            αv = αv - val
            αp = αp - val
            counter = 0
        end
        if gcounter > 350
            println("The simulation has failed to converge.")
            break
        end

    end
    println("Total Iterations: ", gcounter)


    return ustore, vstore, Pstore
end

function AMaker2D(ap, ae, aw, an, as, sizer)
    ny, nx = sizer
    n = nx * ny
    A = zeros(n, n)
    for j in 1:ny
        for i in 1:nx
            k = (j-1)*nx + i
            A[k, k] = ap[j, i]  # Main diagonal
            if i != nx
                A[k, k+1] = -ae[j, i]  # East
            end
            if i != 1
                A[k, k-1] = -aw[j, i]  # West
            end
            if j != ny
                A[k, k+nx] = -an[j, i]  # North
            end
            if j != 1
                A[k, k-nx] = -as[j, i]  # South
            end
        end
    end
    return A
```

```
end



function RichardsonExtrapolation(ConMetricf, Nf)
    P = log((ConMetricf[3] - ConMetricf[2])/ (ConMetricf[4] - ConMetricf[3])) / log(2)
    println("The order of the simulation is $(round(P,digits=2)).")
    QFinal = ConMetricf[4] + (ConMetricf[3] - ConMetricf[4]) / (1 - 2^(round(P,digits=2)))
    println("The grid converged value is $(round(QFinal,digits=2)).")
    return P, QFinal
end

function meshgrid(x, y)
    return repeat(x', length(y)), repeat(y, 1, length(x))
end

#to make this work with different sized u and v and do quiver and stream plots
#=
using Interpolations

# Assuming u is defined on grid xu, yu and v is defined on grid xv, yv
xu = ...
yu = ...
xv = ...
yv = ...

# Create interpolation objects for u and v
interp_u = interpolate((yu, xu), u, Gridded(Linear()))
interp_v = interpolate((yv, xv), v, Gridded(Linear()))

# Define the grid on which you want to plot the vector field
xplot = ...
yplot = ...

# Interpolate u and v onto the plot grid
uplot = interp_u[yplot, xplot]
vplot = interp_v[yplot, xplot]

# Generate a grid of points
X, Y = meshgrid(xplot, yplot)

# Create a quiver plot for the vector field
quiver(X, Y, quiver=(uplot, vplot), color=:grayscale)

# Create a streamline plot
streamplot(X, Y, uplot, vplot, color=:grayscale)
=#
```

FinalProjectTwoDCoupledFlowSolver\TwoDCoupledFlowSolver.jl

```julia
#=
TwoDCoupledFlowSolver.jl
Jacob Child
April 11th, 2024
High Level Psuedocode: implement the SIMPLE CFD algorithm and solve for 2D channel flow
Problem: A 2D channel flow is to be solved using the SIMPLE CFD algorithm. Use an upwind scheme backward staggered grid with five pressure nodes and four velocity nodes. The stagnation pressur eis given at the inlet and the static pressure is specified at the exit.
Assumptions: steady and viscous and the density of the fluid is constant.

Steps To Solve: Discritize, initialize, then calculate the new u using the A and b matrix maker and solver, then do the same for the pressure.

=#
using Plots
include("TwoDSimpleFuncs.jl")

# Givens
ρ = 1000 #kg/m^3
μ = 0.001 #Pa*s
L = 5 / 100 #m
W = 1 / 100 #m
#Boundary Conditions, no slip walls on the top and bottom
Vin = 0.001 #m/s
Pe = 0 #Pa
mdotin = ρ * Vin * W
#Initial guesses
u = 0.001 #m/s
v = 0.0001 #m/s
P  = 0.001 #Pa
#Under Relaxation Factors
αu = .5 #under relaxation factor for velocity, .4 kind of works for 64
αv = .5 #under relaxation factor for velocity
αp = 1 #under relaxation factor for pressure

#Discritize, 4x4 interior Pressure nodes, 5x4 interior u, 4x3 interior v
N = 64
P = ones(N,N) * P
u = ones(N,N+1) * u
v = ones(N+1,N+2) * v
dx = L/N
dy = W/N

#=
#just iterate for a few times in a for loop
us, vs, Ps = [], [], []
push!(us, u)
push!(vs, v)
push!(Ps, P)
for i in 1:10
    u3, v3, P3 = iterator(us[end],vs[end],Ps[end])
    push!(us, u3)
    push!(vs, v3)
    push!(Ps, P3)
```

```julia
end
=#
#us, vs, Ps = Converger(u,v,P)

#plot prep
Pgx = range(dx/2,L-dx/2,step=dx)
Pgy = range(dy/2,W-dy/2,step=dy)
ugx = range(0, L, step=dx)
ugy = range(dy/2, W-dy/2, step=dy)
vgx = range(0-dx/2, L+dx/2, step=dx)
vgy = range(0, W, step=dy)
#for plotting
Pxcoords = repeat(Pgx, length(Pgy))
Pycoords = repeat(Pgy, inner = length(Pgx))
ucoordsx = repeat(ugx, length(ugy))
ucoordsy = repeat(ugy, inner = length(ugx))
vcoordsx = repeat(vgx, length(vgy))
vcoordsy = repeat(vgy, inner = length(vgx))
channelx = [0,0,L,L,0]
channely = [0,W,W,0,0]
GeoPlot = plot(channelx,channely,label = "channel", title="Geometry Discritization Plot")
scatter!(Pxcoords, Pycoords, label = "Pressure Nodes", markershape=:star5)
scatter!(ucoordsx, ucoordsy, label = "u Nodes", color =:red)
scatter!(vcoordsx, vcoordsy, label = "v Nodes",markershape =:cross, color =:black)
#plot the results, put us, vs, and Ps on individual contour plots
#final u plot in a filled contour plot including the channel
uPlot = contourf(ugx, ugy, us[end], title="u velocity Contour Plot (N = $N)", color=:viridis, xlabel="x (m)", ylabel="y (m)", aspect_ratio=:equal)
# scatter!(Pxcoords, Pycoords, label = "Pressure Nodes", markershape=:star5)
# scatter!(ucoordsx, ucoordsy, label = "u Nodes", color =:red)
# scatter!(vcoordsx, vcoordsy, label = "v Nodes",markershape =:cross, color =:black)
#final v plot in a filled contour plot including the channel
vPlot = contourf(vgx, vgy, vs[end], title="v velocity Contour Plot(N = $N)", color=:viridis, xlabel="x (m)", ylabel="y (m)", aspect_ratio=:equal)
#final P plot in a filled contour plot including the channel
PPlot = contourf(Pgx, Pgy, Ps[end], title="Pressure Contour Plot (N = $N)", color=:viridis, xlabel="x (m)", ylabel="y (m)", aspect_ratio=:equal)


#plot of pressure vs x along the centerline
midpoints = Int64.([N/2, N/2 + 1]) #assuming always an even N
Pcl = sum(Ps[end][midpoints,:],dims=1) / 2
PclPlot = plot(Pgx, vec(Pcl), title="Pressure vs x along the centerline (N = $N)", xlabel="x (m)", ylabel="Pa", label = "Pressure");

savefig(GeoPlot,"FinalProjectTwoDCoupledFlowSolver/GeoPlot.png")
savefig(uPlot,"FinalProjectTwoDCoupledFlowSolver/uPlot.png")
savefig(vPlot,"FinalProjectTwoDCoupledFlowSolver/vPlot.png")
savefig(PPlot,"FinalProjectTwoDCoupledFlowSolver/PPlot.png")
savefig(PclPlot,"FinalProjectTwoDCoupledFlowSolver/PclPlot.png")
writedlm("FinalProjectTwoDCoupledFlowSolver/Final64Pressure.csv", Ps[end], ',')
writedlm("FinalProjectTwoDCoupledFlowSolver/Final64U.csv", us[end], ',')
writedlm("FinalProjectTwoDCoupledFlowSolver/Final64V.csv", vs[end], ',')
writedlm("FinalProjectTwoDCoupledFlowSolver/Final64Pcl.csv", Pcl, ',')

PclCompPlot = plot(Pgx, vec(Pcl), title="Pressure vs x along the centerline", xlabel="x (m)", ylabel="Pa", label = "Coded 2D Flow Solver Pressure (N=64)");
```

```julia
plot!(CenterlinePressure01cfd[:,1],CenterlinePressure01cfd[:,2], label = "CFD Pressure")
savefig(PclCompPlot,"FinalProjectTwoDCoupledFlowSolver/PclCompPlot.png")

#plot of u velocity at the midline vs y
midline = Int64(N/2)
uMidline = us[end][:, midline]
uMidlineCompPlot = plot(ugy, uMidline, title="u Velocity at the midline (x = 0.025m)", xlabel="y (m)", ylabel="m/s", label = "Coded 2D Flow Solver u Velocity (N=64)");
plot!(uVlmidline01cfd[:,1],uVlmidline01cfd[:,2], label = "CFD u Velocity", legend= :bottomright)
savefig(uMidlineCompPlot,"FinalProjectTwoDCoupledFlowSolver/uMidlineCompPlot.png")

#plot of v velocity at 0.0015 above the bottom wall vs x
ProfileLoc = argmin(abs.(vgy .- 0.0015))
vProfile64 = vs[end][end-ProfileLoc,:]
vProfileCompPlot = plot(vgx, vProfile64, title="v Velocity at 0.0015 above the bottom wall", xlabel="x (m)", ylabel="m/s", label = "Coded 2D Flow Solver v Velocity (N=64)");
plot!(Vlp01[:,1],Vlp01[:,2], label = "CFD v Velocity")
savefig(vProfileCompPlot,"FinalProjectTwoDCoupledFlowSolver/vProfileCompPlot.png")


#!see the next line
throw("stop here")



#Grid independence study. Compare pressure vs x along centerline, and v velocity profile
ProfileLoc = argmin(abs.(vgy .- 0.0015)) #ie whatever node is closest to 0.001
#initialize
Pcls = []
vProfiles = []
uProfiles = []
for N in [4, 8, 16, 32]
    u = 0.001 #m/s
    v = 0.0001 #m/s
    P  = 0.001 #Pa
    #Under Relaxation Factors
    global αu = .5 #under relaxation factor for velocity
    global αv = .5 #under relaxation factor for velocity
    global αp = 1 #under relaxation factor for pressure
    dx = L/N
    dy = W/N
    P = ones(N,N) * P
    u = ones(N,N+1) * u
    v = ones(N+1,N+2) * v
    us1, vs1, Ps1 = Converger(u,v,P)
    midpoints = Int64.([N/2, N/2 + 1]) #assuming always an even N
    push!(Pcls, sum(Ps1[end][midpoints,:],dims=1) / 2)
    midline1 = Int64(N/2)
    push!(uProfiles, us1[end][:, midline1])
    vgy1 = range(0, W, step=dy)
    ProfileLoc1 = argmin(abs.(vgy1 .- 0.0015))
    push!(vProfiles, vs1[end][end-ProfileLoc1,:])
end
# push!(Pcls, Pcl)
# push!(uProfiles, uMidline)
# push!(vProfiles, vProfile64)
```

```
PclConvPlot = plot(title="Convergence study of centerline pressure", xlabel="m", ylabel="Pa")
#for loop isn't working, so I will just do it manually
dx = L/4
Pgx = range(dx/2,L-dx/2,step=dx)
plot!(Pgx, vec(Pcls[1]), label = "N = 4")
dx = L/8
Pgx = range(dx/2,L-dx/2,step=dx)
plot!(Pgx, vec(Pcls[4]), label = "N = 8")
dx = L/16
Pgx = range(dx/2,L-dx/2,step=dx)
plot!(Pgx, vec(Pcls[5]), label = "N = 16")
dx = L/32
Pgx = range(dx/2,L-dx/2,step=dx)
plot!(Pgx, vec(Pcls[6]), label = "N = 32")
dx = L/64
Pgx = range(dx/2,L-dx/2,step=dx)
plot!(Pgx, vec(Pcls[7]), label = "N = 64")
plot!(title = "Convergence Study of Centerline Pressure", xlabel="x (m)", ylabel="Pa")
savefig(PclConvPlot,"FinalProjectTwoDCoupledFlowSolver/PclConvPlot.png")

vProfileConvPlot = plot(title="convergence study of v velocity Profile", xlabel="m/s",
ylabel="m/s")
#for loop isn't working, so I will just do it manually
dx = L/4
vgx = range(0-dx/2, L+dx/2, step=dx)
plot!(vgx, vec(vProfiles[1]), label = "N = 4")
dx = L/8
vgx = range(0-dx/2, L+dx/2, step=dx)
plot!(vgx, vec(vProfiles[2]), label = "N = 8")
dx = L/16
vgx = range(0-dx/2, L+dx/2, step=dx)
plot!(vgx, vec(vProfiles[3]), label = "N = 16")
dx = L/32
vgx = range(0-dx/2, L+dx/2, step=dx)
plot!(vgx, vec(vProfiles[4]), label = "N = 32")
dx = L/64
vgx = range(0-dx/2, L+dx/2, step=dx)
plot!(vgx, vec(vProfiles[5]), label = "N = 64")
plot!(title = "Convergence Study of v Velocity Profile", xlabel="x (m)", ylabel="m/s")
savefig(vProfileConvPlot,"FinalProjectTwoDCoupledFlowSolver/vProfileConvPlot.png")

uProfileConvPlot = plot(title="Convergence Study of u Velocity Profile", xlabel="y (m)",
ylabel="m/s")
#for loop isn't working, so I will just do it manually
dy = W/4
ugy = range(dy/2, W-dy/2, step=dy)
plot!(ugy, vec(uProfiles[1]), label = "N = 4")
dy = W/8
ugy = range(dy/2, W-dy/2, step=dy)
plot!(ugy, vec(uProfiles[2]), label = "N = 8")
dy = W/16
ugy = range(dy/2, W-dy/2, step=dy)
plot!(ugy, vec(uProfiles[3]), label = "N = 16")
dy = W/32
ugy = range(dy/2, W-dy/2, step=dy)
plot!(ugy, vec(uProfiles[4]), label = "N = 32")
dy = W/64
```

```
ugy = range(dy/2, W-dy/2, step=dy)
plot!(ugy, vec(uProfiles[5]), label = "N = 64")
plot!(title = "Convergence Study of u Velocity Profile", xlabel="y (m)", ylabel="m/s")
savefig(uProfileConvPlot,"FinalProjectTwoDCoupledFlowSolver/uProfileConvPlot.png")

for (i, (n, pcl)) in enumerate(zip([4, 8, 16, 32, 64], Pcls))
    dx = L/n
    Pgx = range(dx/2,L-dx/2,step=dx)
    plot!(Pgx, vec(pcl), label = "N = $(2^(i+1))")
end
#find the maximum value in each row of Pcl
maxPcl = [maximum(row) for row in Pcls]
PclPercentdiff = [abs.(maxPcl[i] - maxPcl[i+1]) / maxPcl[i+1] * 100 for i in 1:length(maxPcl)
-1]

#find the maximum value in each row of uProfiles
maxuProfiles = [maximum(row) for row in uProfiles]
uProfilePercentdiff = [abs.(maxuProfiles[i] - maxuProfiles[i+1]) / maxuProfiles[i+1] * 100
for i in 1:length(maxuProfiles)-1]

#find the maximum value in each row of vProfiles
maxvProfiles = [maximum(row) for row in vProfiles]
vProfilePercentdiff = [abs.(maxvProfiles[i] - maxvProfiles[i+1]) / maxvProfiles[i+1] * 100
for i in 1:length(maxvProfiles)-1]
```

FinalProjectTwoDCoupledFlowSolver\TwoDValidationHelp.jl

```julia
#=
TwoDValidationHelp.jl
Jacob Child
April 24th, 2024
Pseudocode: some quick calculations to help set up the star ccm validation of my 2D flow solver.
=#

using Plots, DelimitedFiles

h = 1/ 100 #m
l = 5/100 #m
ρ = 1000 #kg/m^3
μ = 0.001 #Pa*s
Vin = 0.001 #m/s
Pe = 0 #Pa

Re = ρ * Vin * l / μ #Re = 50, very low, safe to assume laminar

#Prism layer calculations
r = 1.2 #growth rate
H = 1.72*l/sqrt(Re) #blasius δstar for laminar flow, instead of schlicting for turbulent
#assume y+ = 1
cf = .664/sqrt(Re) #blasius for laminar flow, instead of schlicting for turbulent
h = l/Re *sqrt(2/cf) #height of the first cell
n = log(r,H/h*(r-1)+1) #number of cells, 14, (13.415)

#Convergence values
B = [.01,.02,.04,.08] #Base size
Pcl = [.005945, .005949, .005915, .005896] #Pressure at centerline entrance
Umc = [.001496, .001495, .001490, .001486] #u velocity at mid and centerline
Vcle = [-.000010469, .000001125, .000009025, .0000066756] #v velocity at centerline exit
#Percent changes between values with the value before
PclPercChange = [(Pcl[i]-Pcl[i-1])/Pcl[i-1] * 100 for i in 2:length(Pcl)]
UmcPercChange = [(Umc[i]-Umc[i-1])/Umc[i-1] * 100 for i in 2:length(Umc)]
VclePercChange = [(Vcle[i]-Vcle[i-1])/Vcle[i-1] * 100 for i in 2:length(Vcle)]



#Plots
PercChangePlot = plot(B[1:end-1],PclPercChange, label="Pressure Centerline Percent Change", xlabel="Base Size", ylabel="Percent Change", title="Percent Change")
plot!(B[1:end-1],UmcPercChange, label="U Mid Centerline Percent Change")
#plot!(B[1:end-1],VclePercChange, label="V Centerline Exit Percent Change")

#compare v velocity at centerline profiles instead of specific value
Vcle01, header = readdlm("J:\\ComputationalFluids541\\CenterlineVVelocity01.csv", header = true, ',')
Vcle02, header = readdlm("J:\\ComputationalFluids541\\CenterlineVVelocity02.csv", header = true, ',')
Vcle04, header = readdlm("J:\\ComputationalFluids541\\CenterlineVVelocity04.csv", header = true, ',')
Vcle08, header = readdlm("J:\\ComputationalFluids541\\CenterlineVVelocity08.csv", header = true, ',')
VclePlot = plot(Vcle01[:,1],Vcle01[:,2], label="Base Size 0.01")
```

```julia
plot!(Vcle02[:,1],Vcle02[:,2], label="Base Size: 0.02m")
plot!(Vcle04[:,1],Vcle04[:,2], label="Base Size: 0.04m")
plot!(Vcle08[:,1],Vcle08[:,2], label="Base Size: 0.08m")
plot!(title = "Centerline v Velocity Profiles", xlabel = "x", ylabel = "v (m/s)")
#The above does not show any convergence, I am going to repeat, but not at the centerline, instead at 0.0015 above the bottom wall
#TODO: readdlm and plot the v velocity at 0.0015 above the bottom wall for the four base sizes
#TODO: export all of the u, v, P, contour plots to the to my doc
Vlp01, header = readdlm("J:\\ComputationalFluids541\\VVelocity0015NearEdge01.csv", header = true, ',')
Vlp02, header = readdlm("J:\\ComputationalFluids541\\VVelocity0015NearEdge02.csv", header = true, ',')
Vlp04, header = readdlm("J:\\ComputationalFluids541\\VVelocity0015NearEdge04.csv", header = true, ',')
Vlp08, header = readdlm("J:\\ComputationalFluids541\\VVelocity0015NearEdge08.csv", header = true, ',')
VlpPlot = plot(Vlp01[:,1],Vlp01[:,2], label="Base Size 0.01")
plot!(Vlp02[:,1],Vlp02[:,2], label="Base Size: 0.02m")
plot!(Vlp04[:,1],Vlp04[:,2], label="Base Size: 0.04m")
plot!(Vlp08[:,1],Vlp08[:,2], label="Base Size: 0.08m")
plot!(title = "v Velocity Profiles at 0.0015 above the bottom wall", xlabel = "x", ylabel = "v (m/s)")

#TODO: Pcl plot from 0.01 grid size to compare
uVlmidline01cfd, header = readdlm("J:\\ComputationalFluids541\\UVelocMidline01.csv", header = true, ',')
CenterlinePressure01cfd, header = readdlm("J:\\ComputationalFluids541\\CenterlinePressure01.csv", header = true, ',')
```