**HW 7 SIMPLE Algorithm**

A graph of a pressure and position

Description automatically generated**Jacob Child**

A graph with a line

Description automatically generated

**Summary:** 1D flow through a converging nozzle was solved using the SIMPLE CFD algorithm. The example in the textbook on page 200 was followed.

**Assumptions:** constant density, frictionless, steady state, 1D

**Approach:** using the example as a guide, implement the SIMPLE method with an upwind differencing scheme.

**Discussion/Reflection:** As can be seen in the pressure vs position plot, the SIMPLE algorithm does wonderfully at predicting the solution to this problem. 200 grid points was enough to be quite closely converged to the proper mass flow rate value as well (0.22% over). I learned several coding things that were very helpful, however my conceptual understand of what is actually happening and the boundary conditions is lacking a bit, and I will need to more closely read the example and notes on the SIMPLE algorithm. It was found that modifying the under relaxation factor was critical. When I first did 32 control volumes, it wouldn’t converge, and I had to try several values until I settled on 0.05. I didn’t realize how fast the code ran, so I would initially wait a while to see if it would converge, not knowing that if it didn’t stop running in a few seconds it had likely blown up.

#=

SIMPLE1Funcs.jl

Jacob Child

April 3rd, 2024

Psuedocode: functions needed to assist in the SIMPLE CFD algorithm, also includes functions specific to the 2D planar nozzle problem on pg. 200, chp 6 example 6.2

=#

function Area(xf::Union{Float64,Int64}; Aaf = A\_A, Aef = A\_E, Lf = L)

    return Aaf + (Aef - Aaf)\*xf/Lf

end

function Area(xf::AbstractArray; Aaf = A\_A, Aef = A\_E, Lf = L)

    return @. Aaf + (Aef - Aaf)\*xf/Lf

end

"""

uABMaker(uf; Csg = Pg, Cpg = Vg, dxf = dx)

This function creates the A and b matrix for the velocity field.

Inputs:

    uf: velocity field

    Csg: Control surface grid, in this case = pressure grid

    Cpg: Control point grid, in this case = velocity grid

    dxf: grid spacing

"""

function uABMaker(uf, Pf; Csg = Pg, Cpg = Vg, dxf = dx, P0f = P0, Pef = Pe)

    #East and West control surface values needed for the \*internal velocity nodes\* only

    De = 0 #!  both Ds = 0 Just for HW7 problem, no friction

    Dw = 0

    ucs = [(uf[i+1] + uf[i])/2 for i in 1:length(uf)-1]

    Acss = Area(Csg[2:end-1])

    Fs = @. ρ\*Acss\*ucs #F at the control surface, so Fe = Fs[i+1] and Fw = Fs[i-1]

    #println(length(Fs))

    ΔPf = -diff(Pf) #negative because we want left minus right and diff does right index minus left index #? could this cause problems?

    #Initialize Ae, Aw, Ap, b, and d vectors to be put in the matrix later

    Ae = zero(uf)

    Aw = zero(uf)

    Ap = zero(uf)

    b = zero(uf)

    d = zero(uf)

    #Internal node for loop

    for i in eachindex(Fs[1:end-1])

        Ae[i+1] = De + max(-Fs[i+1], 0)

        Aw[i+1] = Dw + max(Fs[i], 0)

        Ap[i+1] = Ae[i+1] + Aw[i+1] + (Fs[i+1] - Fs[i])

        #println("We are subtracting: ", Fs[i+1], " and ", Fs[i])

        b[i+1] = ΔPf[i+1] \* (Acss[i] + Acss[i+1]) / 2

        d[i+1] = Area(Cpg[i+1]) / Ap[i+1]

    end

    #println("Ap: ", Ap)

    #Boundary conditions/external nodes

    #node 1

    uA = uf[1]\*Area(Cpg[1])/Area(Csg[1])

    Fe1 = .5 \* ρ \* (uf[1] + uf[2]) \* Area(Csg[2])

    Fw1 = ρ \* uA \* Area(Csg[1])#?

    #println("Fe1: ", Fe1, " Fw1: ", Fw1)

    Ae[1] = De + max(-Fe1, 0)

    Aw[1] = 0 #Dw + max(Fw1, 0) #? why is this hardcoded to 0?

    Ap[1] = Fe1 + Fw1 \* .5 \* (Area(Cpg[1]) / Area(Csg[1]))^2

    b[1] = (P0f - Pf[2])\*Area(Cpg[1]) + Fw1 \* uf[1] \* (Area(Cpg[1]) / Area(Csg[1]))

    d[1] = Area(Cpg[1]) / Ap[1]

    #node 4

    Fe4 = ρ \* uf[end] \* Area(Cpg[end]) #?this may just be the initial mdot guess = 1kg/s?

    Fw4 = .5 \* ρ \* (uf[end-1] + uf[end]) \* Area(Csg[end-1])

    #println("Fe4: ", Fe4, " Fw4: ", Fw4)

    Ae[end] = De + max(-Fe4, 0)

    Aw[end] = Dw + max(Fw4, 0)

    Ap[end] = Ae[end] + Aw[end] + (Fe4 - Fw4)

    b[end] = (Pf[end-1] - Pef)\*Area(Cpg[end]) #? I have the wrong sign?

    d[end] = Area(Cpg[end]) / Ap[end]

    return Ae, Aw, Ap, b, d

end

"""

TDMASolver: Tridiagonal Matrix Algorithm solver

Inputs:

    a: (generally Ap) #?subdiagonal github copilot said

    b: (generally Ae) #?diagonal

    c: (generally Aw) #?superdiagonal

    d: (generally b), right hand side

"""

function TDMASolver(a,b,c,d)

    n = length(d)

    P = zero(Float64.(d))

    Q = zero(Float64.(d))

    Φ = zero(Float64.(d))

    for i in 1:n

        if i == 1

            P[i] = b[i] / a[i]

            Q[i] = d[i] / a[i]

        else

            P[i] = b[i] / (a[i] - c[i]\*P[i-1])

            Q[i] = (d[i] + c[i]\*Q[i-1]) / (a[i] - c[i]\*P[i-1])

        end

    end

    Φ[n] = Q[n]

    ns = 1:n

    for i in reverse(ns[1:end-1])

        Φ[i] = P[i]\*Φ[i+1] + Q[i]

    end

    return Φ

end

"""

uPCorrection(ustarf, Pstarf, df; Csg = Pg, Cpg = Vg, dxf = dx, P0f = P0, Pef = Pe)

This function corrects the velocity and pressure fields using the pressure correction matrix

Inputs:

    ustarf: velocity field

    Pstarf: pressure field

    df: d vector from uABMaker

Returns:

    ucorrf: corrected velocity field

    Pcorrf: corrected pressure field

"""

function uPCorrection(ustarf, Pstarf, df; Csg = Pg, Cpg = Vg, dxf = dx, P0f = P0, Pef = Pe)

    #Initialize Ae, Festarf, Aw, Fwstarf, Ap, bpf vectors to be put in the matrix later, these are the pressure correction matrices

    Ae = zero(Pstarf)

    Festarf = zero(Pstarf)

    Aw = zero(Pstarf)

    Fwstarf = zero(Pstarf)

    Ap = zero(Pstarf)

    bpf = zero(Pstarf)

    #Internal node for loop

    for i in eachindex(Pstarf[2:end-1]) #I just shortened the range by 2 to do the internal nodes

        Ae[i+1] = ρ \*df[i+1] \* Area(Cpg[i+1])

        Aw[i+1] = ρ \*df[i] \* Area(Cpg[i])

        Fwstarf[i+1] = ρ \* ustarf[i] \* Area(Cpg[i])

        Festarf[i+1] = ρ \* ustarf[i+1] \* Area(Cpg[i+1])

        Ap[i+1] = Ae[i+1] + Aw[i+1]

        bpf[i+1] = Fwstarf[i+1] - Festarf[i+1]

    end

    #Boundary conditions/external nodes

    #PprimeA and PprimeE are both zero (the entrance and exit are given) so,

    Aw[2] = 0

    Ae[end-1] = 0

    #Clean for tdma

    popfirst!(Ae)

    popfirst!(Aw)

    popfirst!(Ap)

    popfirst!(bpf)

    pop!(Ae)

    pop!(Aw)

    pop!(Ap)

    pop!(bpf)

    Ppf = TDMASolver(Ap, Ae, Aw, bpf)

    push!(Ppf, 0) #node 5

    pushfirst!(Ppf, 0) #node 1

    #Correct P

    Pcorrf = Pstarf + Ppf

    #Correct u

    ucorrf = ustarf + df .\* -diff(Ppf)

    #Correct P node 1

    Pcorrf[1] = P0f - .5 \* ρ \* ucorrf[1]^2 \* (Area(Cpg[1]) / Area(Csg[1]))^2

    return ucorrf, Pcorrf

end

function underrelaxation(uoldf, unewf, poldf, pnewf; αuf = αu, αpf = αp)

    unewf = (1-αuf)\*uoldf + αuf\*unewf

    Pnewf = (1-αpf)\*poldf + αpf\*pnewf

    return unewf, Pnewf

end

function iterator(ustartf, Pstartf;Csg = Pg, Cpg = Vg, dxf = dx, P0f = P0, Pef = Pe, αuf = αu, αpf = αp)

    #make all the keyword arguments global in this scope

    Pg = Csg

    Vg = Cpg

    dx = dxf

    P0 = P0f

    Pe = Pef

    αu = αuf

    αp = αpf

    #initialize variables to store

    ustore = Vector{Vector{Float64}}()

    Pstore = Vector{Vector{Float64}}()

    As = []

    bs = []

    ustore = push!(ustore, ustartf)

    Pstore = push!(Pstore, Pstartf)

    Ae1, Aw1, Ap1, b1, d1 = uABMaker(ustartf, Pstartf)

    ustar1 = TDMASolver(Ap1, Ae1, Aw1, b1)

    uc1, Pc1 = uPCorrection(ustar1,Pstartf,d1)

    unew1, Pnew1 = underrelaxation(u,uc1, P, Pc1)

    push!(ustore, unew1)

    push!(Pstore, Pnew1)

    A1 = AMaker(Ap1, Ae1, Aw1)

    push!(As, A1)

    push!(bs, b1)

    #iterate

    while !isapprox(1+maximum(As[end] \* ustore[end-1] - bs[end]), 1, atol = 1e-5)

        Ae, Aw, Ap, b, d = uABMaker(ustore[end], Pstore[end])

        ustar = TDMASolver(Ap, Ae, Aw, b)

        uc, Pc = uPCorrection(ustar,Pstore[end],d)

        unew, Pnew = underrelaxation(ustore[end],uc, Pstore[end], Pc)

        push!(ustore, unew)

        push!(Pstore, Pnew)

        push!(As, AMaker(Ap, Ae, Aw))

        push!(bs, b)

        #to keep As and bs from growing huge

        popfirst!(As)

        popfirst!(bs)

    end

    return ustore, Pstore

end

function AMaker(Apf, Aef, Awf)

    Af = zeros(length(Apf), length(Apf))

    for i in 1:length(Apf)

        Af[i, i] = Apf[i]

        if i != length(Apf)

            Af[i, i+1] = -Aef[i]

        end

        if i != 1

            Af[i, i-1] = -Awf[i]

        end

    end

    return Af

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

#=

SIMPLE1.jl

Jacob Child

April 3rd, 2024

High Level Psuedocode: implement the SIMPLE CFD algorithm and solve for a planar 2D converging nozzle flow

Problem: A planar 2D converging nozzle flow is to be solved using the SIMPLE CFD algorithm. Use the 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 frictionless 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, FLOWMath

include("SIMPLE1Funcs.jl")

# Givens

ρ = 1 #kg/s

L = 2.0 #m

n = 4

dx = L/n #m

A\_A = .5 #m^2

A\_E = .1 #m^2

P0 = 10 #Pa

Pe = 0 #Pa

mdot = 1.0 #kg/s

αu = .8 #under relaxation factor for velocity

αp = .8 #under relaxation factor for pressure

#Discritize

Pg = range(0,L,step=dx)

Vg = range(dx/2,L-dx/2,step=dx)

#Initial Conditions/guess

u = @. mdot/(ρ\*Area(Vg)) #m/s, velocity at each velocity point

P = range(P0, Pe, length = n+1) #Pa, pressure at each pressure point

#Iterate until max(Apstar .\* ustar - bstar) = 0

ustored, Pstored = iterator(u,P)

#prep for convergence study

nalpha = [4,8,16,32,200,256]

prepalphas = [.8, .5, .2, .05, .01,.01]

alphafunc(xf) = linear(nalpha, prepalphas,xf)

#convergence study

function setup(n)

    global dx, Pg, Vg, u, P, αu, αp, mdot

    α = alphafunc(n)

    αp = α

    αu = α

    dx = L/n

    #Discritize

    Pg = range(0,L,step=dx)

    Vg = range(dx/2,L-dx/2,step=dx)

    #Initial Conditions/guess

    u = @. mdot/(ρ\*Area(Vg)) #m/s, velocity at each velocity point

    P = range(P0, Pe, length = n+1) #Pa, pressure at each pressure point

end

mdot4 = ρ \* ustored[end][1] \* Area(Vg[1])

setup(8)

println("n = 8")

ustored, Pstored = iterator(u,P)

mdot8 = ρ \* ustored[end][1] \* Area(Vg[1])

setup(16)

println("n = 16")

ustored, Pstored = iterator(u,P)

mdot16 = ρ \* ustored[end][1] \* Area(Vg[1])

setup(32)

αu = .05

αp = .05

println("n = 32")

ustored, Pstored = iterator(u,P)

mdot32 = ρ \* ustored[end][1] \* Area(Vg[1])

setup(200)

println("n = 200")

ustored, Pstored = iterator(u,P)

mdot200 = ρ \* ustored[end][1] \* Area(Vg[1])

order, Predicted = RichardsonExtrapolation([mdot4, mdot8, mdot16, mdot32],[4,8,16,32])

ConvgPlot = plot([4,8,16,32,200],[mdot4,mdot8,mdot16,mdot32,mdot200],label="mdot vs n",xlabel="n",ylabel="mdot (kg/s)",title="Convergence Study")

hline!([Predicted],label="Richardson Extrapolation Predicted mdot",style=:dash,color=:red)

#Analytical Pressure Solution Plot

#use bernoulli equation and ρ to solve for pressure at each point

mdotana = .44721 #kg/s

Pana = @. P0 - .5\*ρ\*mdotana^2 / ((ρ \* Area(Pg))^2)

PPlot = plot(Pg,Pstored[end],label="Numerical Pressure",xlabel="Position (m)",ylabel="Pressure (Pa)",title="Pressure vs Position")

plot!(Pg,Pana,label="Analytical Pressure",style=:dash)

#=

ns = [4, 8, 16, 32, 64, 128, 256]

mdots = zero(Float64.(ns))

for (i, n) in enumerate(ns)

    global dx, Pg, Vg, u, P, αu, αp, mdot

    α = alphafunc(n)

    αp = α

    αu = α

    dx = L/n

    #Discritize

    Pg = range(0,L,step=dx)

    Vg = range(dx/2,L-dx/2,step=dx)

    #Initial Conditions/guess

    u = @. mdot/(ρ\*Area(Vg)) #m/s, velocity at each velocity point

    P = range(P0, Pe, length = n+1) #Pa, pressure at each pressure point

    un, pn = iterator(u,P)

    mdots[i] = ρ \* un[end][1] \* Area(Vg[1])

end

=#