**ME EN 541 – HW 4**

Handed out 13 February 2024

Due 20 February 2024

|  |  |  |
| --- | --- | --- |
| **Problem** | **Points** | **Topic** |
| **1** | 15 | Convergence criteria parameters |
| **2** | 10 | Successive over-relaxation |
| **3** | 10 | Successive over-relaxation; initial guess |

**Problems 1 to 3 apply to the fin problem described in HW 3.**

1. Explore different convergence criteria parameters by doing the following. Solve for the fin temperature using the conditions of *h* = 10 W/m2K and ** = 0 and Gauss-Seidel iteration. For starting iterations, use an initial uniform temperature of 400 K. *Recall that Gauss-Seidel uses the most recently available value, not the value from the previous iteration.*
   1. Stop iterating when the maximum temperature change between Gauss-Seidel iterations is less than 10−6. Complete the following table.

|  |  |
| --- | --- |
| *Metric: Tip temperature* | |
| CVs | # Iterations |
| 10 | 514 |
| 20 | 1573 |
| 40 | 4952 |
| 80 | 15482 |

* 1. Repeat (a), but instead stop iterating when the maximum tip heat flux change between iterations is less than 10−6. Use the 2nd- and 3rd-to-last nodes for this calculation.

|  |  |
| --- | --- |
| *Metric: Tip heat flux* | |
| CVs | # Iterations |
| 10 | 986 |
| 20 | 3128 |
| 40 | 10283 |
| 80 | 34185 |

* 1. Repeat again, but instead stop iterating when the maximum *base heat flux* change between iterations is less than 10−6.

|  |  |
| --- | --- |
| *Metric: Base heat flux* | |
| CVs | # Iterations |
| 10 | 1061 |
| 20 | 3495 |
| 40 | 12093 |
| 80 | 42674 |

* 1. Comment on your observations. Which measure would you use in practice?

**Comments:**

Tip temperature converges the fastest, and iterations always go up with CVs. I guess that makes sense as the system is becoming less simple, so it will take more iterations to resolve. I feel like tip temperature should be good enough as a convergence measure, however I do wonder if you want to make sure that things have really calmed, in which case using flux is probably a better measure as you want to make sure that the flux isn’t changing. And I guess at the base if you want to be the most accurate.

1. Explore the use of successive over-relaxation (SOR) by doing the following. Solve for the fin temperature using the conditions of *h* = 10 W/m2K and ** = 1. Use the base heat flux as the convergence criteria metric (as in 1c above). Start with an initial guess of *T* = 400 K. Complete the following table with SOR = 1 (regular Gauss-Seidel) and with SOR = 1.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Initial temperature = 400K | | | | |
| CVs | TDMA | Gauss-Seidel | SOR (** = 1.2) | % Reduction |
| 10 | 2 | 1087 | 736 | 32.3 |
| 20 | 2 | 3587 | 2445 | 31.8 |
| 40 | 2 | 12475 | 8469 | 32.1 |
| 80 | 2 | 44083 | 29814 | 32.4 |

1. Repeat the previous problem using the analytical solution for ** = 0 as the initial guess (see HW 3, problem 1).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Initial temperature = Analytical solution with no radiation | | | | |
| CVs | TDMA | Gauss-Seidel | SOR (** = 1.2) | % Reduction |
| 10 | 2 | 1048 | 711 | 32.2 |
| 20 | 2 | 3448 | 2350 | 31.8 |
| 40 | 2 | 11917 | 8150 | 31.6 |
| 80 | 2 | 42060 | 28471 | 32.31 |

#=

HW4.jl

Feb 19th, 2024

Jacob Child

Pseudocode: Do HW3, but use Gauss-Seidel

=#

using Plots

include("HW4Funcs.jl")

#Problem 1

L = .02

D = .003

ϵ = 0

σ = 5.67\*10^-8

K = 401

h = 10

Tb = 400

Tinf = 273

Tsurr = 273

N = 80

Nodes, fw, fe, dx = PracticeB([0,L], N)

#Homogeneous material, so constant K and no qdot

ks = ones(N+2) \* K

Tguess = ones(N+2) \* Tb

T = Tguess

#do a quick iterate 3 times

aw, ap, ae, b = abmaker(Tguess, ks, dx, D, ϵ, σ, h, Tb, Tinf, Tsurr)

#Gauss Seidel by hand first to understand

Test1 = GaussSeidel(Tguess, aw, ap, ae, b)

#Comparer = TDMASolver(ap, ae, aw, b)

#while loop to iterate until error is less than .0001

errors = []

Ts = []

push!(errors, 1)

push!(Ts, Test1)

while errors[end] > 1\*10^-6

Test2 = GaussSeidel(Ts[end], aw, ap, ae, b)

push!(errors, maximum(abs.(Test2 - Ts[end])))

push!(Ts, Test2)

end

println("The number of iterations to get the error below 1\*10^-6 is ", length(errors))

#Redo the iterations, but now with heat flux

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[end-3] - Test1[end-2]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidel(Ts[end], aw, ap, ae, b)

TipHeatFlux = -K \* (Test2[end-2] - Test2[end-1]) / dx

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("The number of iterations to get the heat flux error below 1\*10^-6 is ", length(errors))

#redo but with base heat flux

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[2] - Test1[1]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidel(Ts[end], aw, ap, ae, b)

TipHeatFlux = -K \* (Test2[1] - Test2[2]) / (.5\*dx)

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("The number of iterations to get the base heat flux error below 1\*10^-6 is ", length(errors))

#Problem 2

ϵ = 1;

#N = 20

Nodes, fw, fe, dx = PracticeB([0,L], N)

#Homogeneous material, so constant K and no qdot

ks = ones(N+2) \* K

Tguess = ones(N+2) \* Tb

T = Tguess

#do a quick iterate 3 times

aw, ap, ae, b = abmaker(Tguess, ks, dx, D, ϵ, σ, h, Tb, Tinf, Tsurr)

#Gauss Seidel by hand first to understand

Test1 = GaussSeidelRelaxed(Tguess, aw, ap, ae, b, 1)

#iterate based off of base heat flux convergence

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[2] - Test1[1]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidelRelaxed(Ts[end], aw, ap, ae, b, 1)

TipHeatFlux = -K \* (Test2[1] - Test2[2]) / (.5\*dx)

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("Part 2 The number of iterations to get the base heat flux error below 1\*10^-6 is ", length(errors))

N1 = length(errors)

#Gauss Seidel by hand first to understand

Test1 = GaussSeidelRelaxed(Tguess, aw, ap, ae, b, 1.2)

#iterate based off of base heat flux convergence

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[2] - Test1[1]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidelRelaxed(Ts[end], aw, ap, ae, b, 1.2)

TipHeatFlux = -K \* (Test2[1] - Test2[2]) / (.5\*dx)

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("Part 2 The number of iterations with relaxation to get the base heat flux error below 1\*10^-6 is ", length(errors))

N2 = length(errors)

percentdif = (N2 - N1) / N1 \* 100

println("Part 2 The percent difference between the two is ", percentdif)

#Part 3

n = sqrt( 4\*h / (D \* K))

Tanalytical = @. cosh(n\*(L - Nodes)) / cosh(n\*L) \* (Tb - Tinf) + Tinf

#Gauss Seidel by hand first to understand

Test1 = Tanalytical

#iterate based off of base heat flux convergence

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[2] - Test1[1]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidelRelaxed(Ts[end], aw, ap, ae, b, 1)

TipHeatFlux = -K \* (Test2[1] - Test2[2]) / (.5\*dx)

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("Part 3 The number of iterations to get the base heat flux error below 1\*10^-6 is ", length(errors))

N1 = length(errors)

#Gauss Seidel by hand first to understand

Test1 = Tanalytical

#iterate based off of base heat flux convergence

errors = []

qs = []

Ts = []

push!(Ts, Test1)

push!(errors, 1)

push!(qs, -K \* (Test1[2] - Test1[1]) / dx)

while errors[end] > 1\*10^-6

Test2 = GaussSeidelRelaxed(Ts[end], aw, ap, ae, b, 1.2)

TipHeatFlux = -K \* (Test2[1] - Test2[2]) / (.5\*dx)

push!(errors, abs(qs[end] - TipHeatFlux))

push!(Ts, Test2)

push!(qs, TipHeatFlux)

end

println("Part 3 The number of iterations with relaxation to get the base heat flux error below 1\*10^-6 is ", length(errors))

N2 = length(errors)

percentdif = (N2 - N1) / N1 \* 100

println("Part 3 The percent difference between the two is ", percentdif)

#=

HW3Funcs.jl

=#

"""

Practice B: This function takes in bounds and the number of control volumes. It returns the node locations (length CV + 2), control surface locations (length CV + 1), and dx.

"""

function PracticeB(boundsf::Vector{Float64}, nf::Int)

dx = (boundsf[2] - boundsf[1])/nf #Control volume width from splitting the bounds into n parts

#Practice B puts nodes at the bounds and in the middle of the control volumes

Interiors = range(boundsf[1] + dx/2, boundsf[2] - dx/2, length=nf) #interior node locations

#! The above line is different than the original, but it is more robust and less error prone

#! original: Interiors = boundsf[1] + dx/2:dx:boundsf[2] - dx/2 #interior node locations

Nodes = [boundsf[1]; Interiors; boundsf[2]] #Node locations with the bounds

#CS = boundsf[1]:dx:boundsf[2] #Control surface locations (evenly spaced including the bounds)

CS = range(boundsf[1], boundsf[2], length(Nodes)-1) #? The above causes issues at weird points like N = 44

#calculate fw and fe for each node for the harmonic mean

fw = zeros(length(Nodes))

fe = zeros(length(Nodes))

fw[1] = 0

fe[1] = 1

fw[end] = 1

fe[end] = 0

for i in 2:length(Nodes)-1

fw[i] = (CS[i-1] - Nodes[i-1]) / (Nodes[i] - Nodes[i-1])

fe[i] = (Nodes[i+1] - CS[i]) / (Nodes[i+1] - Nodes[i])

end

return Nodes, fw, fe, dx

end

function HMKNew(kf,fwf, fef)

kw = zeros(length(kf))

ke = zeros(length(kf))

for i in 1:length(kf)

if i == 1

ke[i] = 1 / ((1 - fef[i])/kf[i] + fef[i]/kf[i+1])

kw[i] = 0 # this shouldn't be used 1 / ((1 - fwf[i])/kf[i] + fwf[i]/kf[i-1])

elseif i == length(kf)

ke[i] = 0 #This shouldn't be used 1 / ((1 - fef[i])/kf[i] + fef[i]/kf[i+1])

kw[i] = 1 / ((1 - fwf[i])/kf[i] + fwf[i]/kf[i-1])

else

kw[i] = 1 / ((1 - fwf[i])/kf[i] + fwf[i]/kf[i-1])

ke[i] = 1 / ((1 - fef[i])/kf[i] + fef[i]/kf[i+1])

end

end

return kw, ke

end

function MaterialMaker5(Nodesf, kf, qdotf, MatInterfaceLocf)

k = zero(Nodesf) #Vector of thermal conductivities in the control volumes

qdot = zero(Nodesf) #Vector of heat generation rates in the control volumes

for (i, node) in enumerate(Nodesf)

if node < MatInterfaceLocf

k[i] = kf[1]

qdot[i] = qdotf[1]

else

k[i] = kf[2](node)

qdot[i] = qdotf[2]

end

end

return k, qdot

end

"""

FiniteDif: Finite Difference Function for 1D heat conduction with an adiabatic surface at the left boundary and a convection surface at the right boundary. The goal is to find the aw, ap, ae and b coefficients for each node and then solve the matrix to find the temperatures at each node.

Inputs:

kf: Vector of thermal conductivities in the materials (same length as Nodesf)

qdotf: Vector of heat generation rates in the materials (same length as Nodesf)

hf: Convection coefficient at the right boundary

Tinf: Ambient temperature at the right boundary

dxf: Width of the control volumes

Nodesf: Vector of node locations

CSf: Vector of control surface locations

MatInterfaceLocf: Vector of the locations of the interfaces between materials

"""

function FiniteDif(kwf, kef, qdotf, Nodesf, hf, Tinf, dxf)

#Initialize the vectors for the coefficients

ap = zeros(length(Nodesf))

ae = zeros(length(Nodesf))

aw = zeros(length(Nodesf))

b = zeros(length(Nodesf))

#big for loop to go through the nodes

#always compute the harmonic mean for ke and kw, and use MatInterfaceLoc to know when to switch materials and thus qdot and which k value is east and west etc

for (i, \_) in enumerate(Nodesf)

#for the HMKe and HMKw function keyword arguments

dx = dxf

if i == 1 #Left boundary

ap[i] = 2\*kef[i] / dxf

ae[i] = 2\*kef[i] / dxf

aw[i] = 0

b[i] = 0

elseif i == 2 #Node 2

ap[i] = kef[i] / dxf + 2\*kwf[i] / dxf

ae[i] = kef[i] / dxf

aw[i] = 2\*kwf[i] / dxf

b[i] = qdotf[1]\*dxf

elseif i == length(Nodesf) - 1 #Node 6

ap[i] = 2\*kef[i] /dxf + kwf[i] / dxf

ae[i] = 2\*kef[i] / dxf

aw[i] = kwf[i] / dxf

b[i] = 0

elseif i == length(Nodesf) #Right boundary

ap[i] = 2\*kwf[i] / dxf + hf

ae[i] = 0

aw[i] = 2\*kwf[i] / dxf

b[i] = hf\*Tinf

else #Nodes 3-5, because I used MaterialMaker, I don't need to worry about where the conditions change

ap[i] = kef[i] / dxf + kwf[i] / dxf

#println("At node $i, Ke is $(kf[i+1]) and Kw is $(kf[i-1]), and Kp is $(kf[i]).")

ae[i] = kef[i] / dxf

aw[i] = kwf[i] / dxf

b[i] = qdotf[i]\*dxf

end

end

#Make the A and b Matricies, solve for T

A = zeros(length(Nodesf), length(Nodesf))

for i in 1:length(Nodesf)

A[i, i] = ap[i]

if i != length(Nodesf)

A[i, i+1] = -ae[i]

end

if i != 1

A[i, i-1] = -aw[i]

end

end

T = A\b

return T, A, b

end

function EasyRun5(bounds, N, ka, kb, qdota, h, T∞, MatInterfaceLoc)

x,fw, fe, dx = PracticeB(bounds, N)

#safety check

if length(x) -2 != N

throw("PracticeB did not return the correct number of nodes. N = $N")

end

ks, qdot = MaterialMaker5(x,[ka,kb],[qdota, 0],MatInterfaceLoc)

#safety check

if length(ks) -2 != N

throw("MaterialMaker5 did not return the correct number of materials. N = $N")

end

kw,ke = HMKNew(ks,fw,fe)

NewKw = []

NewKe = []

for i in 1:length(ks)

if i == 1

push!(NewKw, "Kwcatch")

push!(NewKe, ke[i])

elseif i == length(ks)

push!(NewKw, kw[i])

push!(NewKe, "KeCatch")

else

push!(NewKw, kw[i])

push!(NewKe, ke[i])

end

end

T, A, b = FiniteDif(NewKw, NewKe, qdot, x, h, T∞, dx)

return T, A, b

end

###

###

#Functions for HW3 Problems 2-3 TDMA

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

"""

abmaker: function to make the ae, ap, aw, and b vectors given a Tpstar, K values, dx, and all the given constants

"""

function abmaker(Tpstar, Kf, dxf, Df, emf, sigmaf, hf, Tbasef, Tinff, Tsurrf)

n = length(Kf)

ap = zero(Float64.(Kf))

ae = zero(Float64.(Kf))

aw = zero(Float64.(Kf))

b = zero(Float64.(Kf))

#Calculations

Sc = @. 4/Df \*hf\*Tinff + 4\*emf\*sigmaf/Df \* Tsurrf^4 + 12 \* emf \* sigmaf \* Tpstar^4 / Df

Sp = @. -4 \* hf / Df - 16 \* emf \* sigmaf \* Tpstar^3 / Df

for i in 1:n

if i == 1

aw[i] = 0

ae[i] = 0

ap[i] = ae[i] + aw[i] + 1

b[i] = Tbasef

elseif i == 2

aw[i] = 2\*Kf[i] / dxf

ae[i] = Kf[i] / dxf

ap[i] = ae[i] + aw[i] - Sp[i] \* dxf

b[i] = Sc[i] \* dxf

elseif i == n-1

aw[i] = Kf[i] / dxf

ae[i] = 2\*Kf[i] / dxf

ap[i] = ae[i] + aw[i] - Sp[i] \* dxf

b[i] = Sc[i] \* dxf

elseif i == n

#adiabatic tip

aw[i] = 1

ae[i] = 0

ap[i] = ae[i] + aw[i]

b[i] = 0 #? is this right?

else

aw[i] = Kf[i] / dxf

ae[i] = Kf[i] / dxf

ap[i] = ae[i] + aw[i] - Sp[i] \* dxf

b[i] = Sc[i] \* dxf

end

end

return aw, ap, ae, b

end

function GaussSeidel(Toldf, awf, apf, aef, bf)

Tnew = zero(Toldf)

Tnew[1] = bf[1] / apf[1]

Tnew[2] = (awf[2]\*Tnew[1] + aef[2]\*Toldf[3] + bf[2]) / (apf[2])

for i in 3:length(Toldf)-2

Tnew[i] = (awf[i]\*Tnew[i-1] + aef[i]\*Toldf[i+1] + bf[i]) / (apf[i])

end

Tnew[end-1] = (awf[end-1]\*Tnew[end-2] + aef[end-1]\*Toldf[end] + bf[end-1]) / (apf[end-1])

Tnew[end] = (awf[end]\*Tnew[end-1] + aef[end]\*Toldf[end] + bf[end]) / (apf[end])

return Tnew

end

function GaussSeidelRelaxed(Toldf, awf, apf, aef, bf,alphaf)

Tnew = zero(Toldf)

Tnew[1] = bf[1] / apf[1]

Tnew[2] = Toldf[2] + ((awf[2]\*Tnew[1] + aef[2]\*Toldf[3] + bf[2]) / (apf[2]) - Toldf[2]) \* alphaf

for i in 3:length(Toldf)-2

Tnew[i] = Toldf[i] + ((awf[i]\*Tnew[i-1] + aef[i]\*Toldf[i+1] + bf[i]) / (apf[i]) - Toldf[i]) \* alphaf

end

Tnew[end-1] = Toldf[end-1]+((awf[end-1]\*Tnew[end-2] + aef[end-1]\*Toldf[end] + bf[end-1]) / (apf[end-1]) - Toldf[end-1]) \* alphaf

Tnew[end] = Toldf[end] + ((awf[end]\*Tnew[end-1] + aef[end]\*Toldf[end] + bf[end]) / (apf[end]) - Toldf[end]) \* alphaf

return Tnew

end