```julia
# ---------------------------------------------------------------------------
# Problem Set 1 (2019): The Eaton-Korturm (2002) Economy
# Written by Heejin Yoon
# ---------------------------------------------------------------------------

## keyword-enabled structure to hold model primitives

@with_kw struct Primitives
    nk::Int64 = 100000  # number of goods
    nc::Int64 = 3  # number of countries
    θ::Float64 = 4.0  # Frechet parameter
    σ::Float64 = 2.0  # elasticity of substitution
    l::Array{Float64} = ones(nc)  # (inelastic) labor supply
end

## structure that holds model results

mutable struct Results
    z::Array{Float64, 2}  # productivity of each country & each good (100,000 × 3)
    w::Array{Float64, 2}  # wage of each country (1 × 3)
    p::Array{Float64, 2}  # actual price of each good imported to each country (100,000
× 3)
    p_index::Array{Float64}  # CES price index of each country (3 × 1)
    supplier::Array{Int64, 2}  # supplier of each good imported to each country
(100,000 × 3)
    X::Array{Float64, 2}  # bilateral trade flow
    EX::Array{Float64}  # excess trade flow: export flow - import flow of each country
    Π::Array{Float64, 2}  # bilateral trade share
end

## function for initializing model primitives and results

function Initialize()
    prim = Primitives()
    z = zeros(prim.nk, prim.nc)
    w = ones(1, prim.nc)
    p = zeros(prim.nk, prim.nc)
    p_index = zeros(prim.nc)
    supplier = zeros(prim.nk, prim.nc)
    X = zeros(prim.nc, prim.nc)
    EX = zeros(prim.nc)
    Π = zeros(prim.nc, prim.nc)
    res = Results(z, w, p, p_index, supplier, X, EX, Π)
    prim, res
end

## draw productivity of each country

function draw_productivity(prim::Primitives, res::Results, T::Array{Float64})
    @unpack nk, nc, θ = prim

    Random.seed!(1234)
    z = zeros(nk, nc)
    b = T.^(1/θ) # command Frechet(a, b) in Julia gives the distribution
exp(-(x/b)^(-a)) ⇒ a = θ, b = (T_i)^(1/θ)

    for c_index = 1:nc  # country index from 1 to 3
        dist = Frechet(θ, b[c_index])
        for k_index = 1:nk
```

```julia
            z[k_index, c_index] = rand(dist)
        end
    end

    z
end

## calculate the price and the supplier country of each product imported by each
country.

function price_supplier(prim::Primitives, res::Results, τ::Array{Float64, 2})
    @unpack σ, nk, nc = prim
    @unpack w, z = res

    latent_p = zeros(nk, nc, nc)
    p = zeros(nk, nc)
    supplier = zeros(Int64, nk, nc)

    # calculate the prices of goods offered by each country
    for j_index = 1:nc
        for i_index = 1:nc
            for k_index = 1:nk
                latent_p[k_index, i_index, j_index] = w[i_index]/z[k_index, i_index] *
(1 + τ[i_index, j_index])
            end
        end
    end

    # for each importing country, the minimum price offer will be the actual price.
    for j_index = 1:nc
        for k_index = 1:nk
            p[k_index, j_index] = minimum(latent_p[k_index, :, j_index])
            supplier[k_index, j_index] = argmin(latent_p[k_index, :, j_index])
        end
    end

    # calculate the price index using the actual price.
    p_index = zeros(nc)
    for c_index = 1:nc
        p_index[c_index] = (sum(p[:, c_index].^(1-σ)))^(1/(1-σ))
    end

    p, p_index, supplier
end

## calculate the bilateral trade flows and the excess trade flow.

function trade_flow(prim::Primitives, res::Results)
    @unpack σ, nk, nc, l = prim
    @unpack w, p, p_index, supplier = res

    c = zeros(nk, nc)
    expenditure = zeros(nk, nc)
    X = zeros(nc, nc)
    EX = zeros(nc)

    # calculate the demand of importing countries for k goods, and the total importing
value.
    for c_index = 1:nc
```

```
        for k_index = 1:nk
            c[k_index, c_index] = (p[k_index, c_index]/p_index[c_index])^(-σ) *
((w[c_index]*l[c_index])/p_index[c_index])
            expenditure[k_index, c_index] = c[k_index, c_index] * p[k_index, c_index]
        end
    end

    # calculate the bilateral trade flows.
    for j_index = 1:nc
        for i_index = 1:nc
            for k_index = 1:nk
                if supplier[k_index, j_index] == i_index
                    X[i_index, j_index] += expenditure[k_index, j_index]
                end
            end
        end
    end

    # calculate the excess trade flow of each country.
    for i_index = 1:nc
        EX[i_index] = sum(X[i_index, :]) - sum(X[:, i_index])
    end

    X, EX
end

## define find_wage function to minimize the excess trade flow

function find_wage(ww)
    res.w = [1.0 ww]
    res.p, res.p_index, res.supplier = price_supplier(prim, res, τ)
    res.X, res.EX = trade_flow(prim, res)
    err = sum(abs.(res.EX))

    err
end

## solve for the balanced trade condition

function solve_balanced_trade(prim::Primitives, res::Results, τ::Array{Float64, 2})
    @unpack nc = prim

    # find the wage levels of countries 2 and 3, which minimizes the sum of excess
trade flow of three countries.
    w_1 = optimize(find_wage, [1.0 1.0], NelderMead()).minimizer
    res.w = [1.0 w_1]

    # calculate the trade flow share
    for i_index = 1:nc
        for j_index = 1:nc
            res.Π[i_index, j_index] = res.X[i_index, j_index] / (res.w[j_index] *
prim.l[j_index])
        end
    end

end

## welfare gain obtained by moving from autarky
```

```
function welfare_gain(prim::Primitives, res::Results)
    @unpack σ, nk, nc = prim
    @unpack z, w, p_index = res

    w_autarky = ones(nc, 1)
    latent_p_autarky = zeros(nk, nc, nc)
    p_index_autarky = zeros(nc, 1)

    for j_index = 1:nc
        for i_index = 1:nc
            for k_index = 1:nk
                latent_p_autarky[k_index, i_index, j_index] =
w_autarky[i_index]/z[k_index, i_index]
            end
        end
    end

    for c_index = 1:nc
        p_index_autarky[c_index] = (sum(latent_p_autarky[:, c_index,
c_index].^(1-σ)))^(1/(1-σ))
    end

    wg = zeros(nc)
    for c_index = 1:nc
        wg[c_index] =
(w[c_index]/p_index[c_index])/(w_autarky[c_index]/p_index_autarky[c_index])
    end

    wg
end

## alternative way to calculate the balanced trade condition: adjust w_2 and w_3 little
by little

function solve_balanced_trade_alt(prim::Primitives, res::Results, τ::Array{Float64, 2})
    @unpack nc = prim
    tol = 0.0001
    err = 1.0
    n = 1 # count

    while true
        res.p, res.p_index, res.supplier = price_supplier(prim, res, τ)
        res.X, res.EX = trade_flow(prim, res) #spit out excess trade flows
        err = maximum(abs.(res.EX[2:nc])) #reset error level
        c_index = argmax(abs.(res.EX[2:nc])) + 1
        res.w[c_index] = res.w[c_index] + res.EX[c_index] * 0.1

        println("***** ", n, "th iteration *****")
        println(res.EX)
        @printf("Excess Trade Balance: %0.5f \n", float(err))
        println("***************************")
        n += 1
        if err < tol #begin iteration
            break
        end
    end

    for i_index = 1:nc
        for j_index = 1:nc
```

```
            res.Π[i_index, j_index] = res.X[i_index,
j_index]/res.w[j_index]*prim.l[j_index]
        end
    end
    println("Trade is balanced after ", n, " iterations.\n")

end
```

```
Error: LoadError: UndefVarError: @with_kw not defined
in expression starting at C:\Users\hyoon76\OneDrive - UW-Madison\3.Wisconsi
n\2022 Fall\Econ871\Problem Sets\PS1_2019\ps1_2019_model_hj.jl:9
```