THE PRIMAL SIMPLEX ALGORITHM

David F. Gleich

February 25, 2025

The material here is from Chapter 13 in Nocedal and Wright, but some of the geometry comes from Griva, Sofer, and Nash.

Recall the standard form for a linear program:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0. \end{aligned}$$

```
type SimplexState
    c::Vector
    A::Matrix
    b::Vector
    bset::Vector{Int} # columns of the BFP
end
type SimplexPoint
    x::Vector   #
    binds::Vector{Int}
    ninds::Vector{Int}
    lam::Vector # equality Lagrange mults
    sn::Vector  # non-basis Lagrange mults
    B::Matrix   # the set of basic cols
    N::Matrix   # the set of non-basic cols
end

function SimplexPoint(T::Type, B::Matrix, N::Matrix)
    return SimplexPoint(zeros(T,0),zeros(Int,0),zeros(Int,0),
        zeros(T,0), zeros(T,0), B, N)
end



function simplex_point(s::SimplexState)
    binds = state.bset # basic variable indices
    ninds = setdiff(1:size(A,1),binds) # non-basic
    B = state.A[:,binds]
    N = state.A[:,ninds]
    cb = state.c[binds]
    cn = state.c[ninds]

    if rank(B) != m
        return (:Infeasible, SimplexPoint(eltype(c), B, N))
    end

    xb = B\b
    x = zeros(eltype(xb),n)
    x[binds] = xb
    x[ninds] = zero(eltype(xb))

    lam = B'\cb
    sn = cn - N'*lam

    if any(xb .< 0)
        return (:Infeasible, SimplexPoint(x, binds, ninds, lam, sn, B, N))
    else
        if all(sn .>= 0)
            return (:Solution, SimplexPoint(x, binds, ninds, lam, sn, B, N))
        else
            return (:Feasible, SimplexPoint(x, binds, ninds, lam, sn, B, N))
        end
    end
end

```

```
function simplex_step!(state::SimplexState)
    stat,p::SimplexPoint = simplex_point(state)

    if stat == :Solution
        return stat, p
    elseif state == :Infeasible
        return :Breakdown, p
    else # we have a BFP
        #= This is the Simplex Step! =#

        # take the Dantzig index to add to basic
        qn = indmin(p.sn)
        q = p.ninds[qn] # translate index
        # check that nothing went wrong
        @assert all(state.A[:,q] == p.N[:,qn])

        d = p.B \ state.A[:,q]
        #@show d

        # TODO, implement an anti-cycling method /
        # check for stagnation and lack of progress
        # this checks for unbounded solutions
        if all(d .<= eps(eltype(d)))
            return :Unbounded, p
        end

        # determine the index to remove
        xq = p.x[p.binds]./d
        xq[d .< eps(eltype(xq))] = Inf
        pb = indmin(xq)
        pind = p.binds[pb] # translate index

        #@show p.binds, pb, pind, state.bset, q

        # remove p and add q
        @assert state.bset[pb] == pind
        state.bset[pb] = q

        return stat, p
    end
end

function runsimplex(state::SimplexState)
    @show state
    status, p = simplex_step!(state)
    iter = 1
    while status != :Solution
        @show state.bset
        @show p.x
        status, p = simplex_step!(state)
        iter += 1
    end
    return p, state
end
function simplex_init(c,A,b)
    # need to get an initial BFP
    # setup the LP for phase 1
    m,n = size(A)
    e = sign.(b)
    cc = [zeros(n); ones(m)]
    CA = [A Diagonal(e)]
    cb = b
    bset = collect(n+1:n+m) # the last columns of cc
    phase1 = SimplexState(cc,CA,cb,bset)
    p, state = runsimplex(phase1)
end
```