THE PRIMAL SIMPLEX ALGORITHM

David F. Gleich

February 24, 2026

Again, let me reiterate the importance of this result. We can now use the fact that sets of m indicies of the matrix $\boldsymbol{A}$ give rise to vertices of the feasible polytope.

Overview of Simplex Method The simplex method, then, moves from vertex to vertex on the feasible polytope by adding and subtracting indicies from the subset $\mathcal{B}$ to move between vertices. At every stage, it moves to another vertex with a non-increasing objective.

```
"Presolve" (Remove redundant rows from A)
"Phase 1" Find a feasible vertex / basic point
"Simplex" Find the dual variables / Lagrange multipliers for that point
While KKT conditions are not true
  Move to another point
  Find the dual variables / Lagrange multipliers for new point
```

The first step of finding a feasible vertex or basic point is called a Phase 1 solution. We'll cover that soon.

Columns to basic feasible points and Lagrange multipliers What we now need to address is how to find the dual variables or Lagrange multipliers at a basic feasible point (vertex). *If* we can get these (and we can) then we can determine when to stop the simplex method.

See Section 13.3 for more about this next piece.

Let $\mathbf{x}$ be a basic feasible point. Let $\boldsymbol{B}$ be the subset of columns at any basic feasible point. Thne $\boldsymbol{B}\mathbf{x}_B = \mathbf{b}$ and we can permute $\mathbf{x}$ into $\begin{bmatrix}\mathbf{x}_B^T & \mathbf{x}_N^T\end{bmatrix}^T$, where $\mathbf{x}_B$ are non-zero elements ("the basic elements") of $\mathbf{x}$ and $\mathbf{x}_N = 0$ be the zero elements ("the non-basic elements"). Partition $\mathbf{s}$ (multipliers) and $\mathbf{c}$ (objective) conformally into $\mathbf{s}_B, \mathbf{s}_N$ and $\mathbf{c}_B, \mathbf{c}_N$.

We can write:

$$\boldsymbol{A}\mathbf{x} = \boldsymbol{B}\mathbf{x}_B + \boldsymbol{N}\mathbf{x}_N = \mathbf{b}$$

where $\mathbf{x}_B = \boldsymbol{B}^{-1}\mathbf{b}$ and $\mathbf{x}_B \geq 0$. We set $\mathbf{s}_B = 0$ because all the elements of $\mathbf{x}_B$ handle that portion of the constraint. We now use the remaining KKT conditions to find $\boldsymbol{\lambda}$ and $\mathbf{s}_N$. Note that

$$\boldsymbol{B}^T\boldsymbol{\lambda} = \mathbf{c}_B \text{ and } \boldsymbol{N}^T\boldsymbol{\lambda} + \mathbf{s}_N = \mathbf{c}_N.$$

The first equation defines $\boldsymbol{\lambda}$ because $\boldsymbol{B}^T$ is non-singular. The second equation can then be solved for $\mathbf{s}_N$. Consequently, we can find the dual variables for a solution. *However*, $\mathbf{s}$ need not be non-negative.

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

```
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
```

```
21
22  function simplex_point(s::SimplexState)
23      binds = state.bset # basic variable indices
24      ninds = setdiff(1:size(A,1),binds) # non-basic
25      B = state.A[:,binds]
26      N = state.A[:,ninds]
27      cb = state.c[binds]
28      cn = state.c[ninds]
29
30      if rank(B) != m
31          return (:Infeasible, SimplexPoint(eltype(c), B, N))
32      end
33
34      xb = B\b
35      x = zeros(eltype(xb),n)
36      x[binds] = xb
37      x[ninds] = zero(eltype(xb))
38
39      lam = B'\cb
40      sn = cn - N'*lam
41
42      if any(xb .< 0)
43          return (:Infeasible, SimplexPoint(x, binds, ninds, lam, sn, B, N))
44      else
45          if all(sn .>= 0)
46              return (:Solution, SimplexPoint(x, binds, ninds, lam, sn, B, N))
47          else
48              return (:Feasible, SimplexPoint(x, binds, ninds, lam, sn, B, N))
49          end
50      end
51  end
52
53  function simplex_step!(state::SimplexState)
54      stat,p::SimplexPoint = simplex_point(state)
55
56      if stat == :Solution
57          return stat, p
58      elseif state == :Infeasible
59          return :Breakdown, p
60      else # we have a BFP
61          #= This is the Simplex Step! =#
62
63          # take the Dantzig index to add to basic
64          qn = indmin(p.sn)
65          q = p.ninds[qn] # translate index
66          # check that nothing went wrong
67          @assert all(state.A[:,q] == p.N[:,qn])
68
69          d = p.B \ state.A[:,q]
70          #@show d
71
72          # TODO, implement an anti-cycling method /
73          # check for stagnation and lack of progress
74          # this checks for unbounded solutions
75          if all(d .<= eps(eltype(d)))
76              return :Unbounded, p
77          end
78
79          # determine the index to remove
80          xq = p.x[p.binds]./d
81          xq[d .< eps(eltype(xq))] = Inf
82          pb = indmin(xq)
83          pind = p.binds[pb] # translate index
84
85          # remove p and add q
86          @assert state.bset[pb] == pind
87          state.bset[pb] = q
88
89          return stat, p
90      end
91  end
```