Optimization Conscious Econometrics pset 4

Timothy Schwieg

1 3.18

Consider the Simplex method applied to a standard form problem and assume that the rows of the matrix A are linearly independent. For each of the statements that follow, give either a proof or counterexample.

1. An iteration of the simplex method may move the feasible solution by a positive distance while leaving the cost unchanged.

   In the second step of the simplex method, the reduced costs of moving in every direction is computed. For the cost to be unchanged, the reduced cost in the direction chosen must be zero. However the algorithm terminates if they are all non-negative, and otherwise chooses a strictly negative element of the reduced cost. This means that it will never choose a negative reduced cost.

   Note that if it does not move a positive distance (i.e. it is degenerate), this will leave the cost unchanged.

2. A variable that has just left the basis cannot reenter in the very next iteration.

   For a variable to re-enter the basis, it would require that its reduced cost be negative. However, when a variable leaves the basis, that is because we traveled in a direction that it was the first to hit 0. Thus we increased the objective by decreasing the value of x_ℓ. It cannot be that increasing the value of x_ℓ from that position somehow decreases the value of the objective function. We must move to a different index for this to be possible. That is, it can possibly be up for re-entry on any iteration except the one where it was immediately removed.

3. A variable that has just entered the basis cannot leave in the very next iteration.

   This is false. Consider a simplex determined by a right triangle at the origin. If the reduced cost is negative in any movement from the origin, it is possible that it moves from the origin to the right-most point, then to the top-most point. In this, the x dimension enters the basis, then immediately leaves in favor of y.

2 4.4

Let A be a symmetric square matrix. Consider the linear programming problem:

$$\begin{aligned} \min \quad & c'x \\ \text{s.t.} \quad & Ax \geq c \\ & x \geq 0 \end{aligned}$$

Prove that if x^* satisfies $Ax^* = c$ and $x^* \geq 0$ then x^* is an optimal solution.

The objective function evaluated at x^* is: $x^{*\prime}A'x^*$. Assume that this is not optimal for the linear program. Let y be optimal for the problem. Since $y \neq x^*$ then $Ay \geq c$ with the inequality being strict in at least one component. This can be written as $Ay = c + \epsilon$ where $\epsilon \geq 0$, and it is strict in at least one component.

From the definition of y being a minimizer and $y \neq x^*$ and using $c = Ax^*$: and the symmetry of A:

$$\begin{aligned} c'x^* &> c'y \\ x^{*\prime}A'x^* &> x^{*\prime}A'y \\ x^{*\prime}Ax^* &> x^{*\prime}Ay \\ x^{*\prime}c &> x^{*\prime}(c+\epsilon) \\ 0 &> x^{*\prime}\epsilon \end{aligned}$$

However we know that $x^* \geq 0$ and $\epsilon \geq 0$, so it must be the case that $x^{*\prime}\epsilon \geq 0$. This is a contradiction, and therefore we find that $y = x^*$. Thus x^* must be optimal.

3 4.26

Show that exactly one of the following holds:

1. There exists an $x \neq 0$ such that $Ax = 0, x \geq 0$
2. There exists some p such that $p'A > 0$

Examine condition (1). For there to be a vector $x \neq 0$ such that $Ax = 0$, then the null space of A must have dimension greater than zero. This means that some column can be written as a linear combination of the previous columns, but even stronger than this, we require that the weights used in the linear combination to be non-negative.

Let $a_1, ..., a_m$ denote the columns of A, then this means that $c_1a_1 + ... + c_ma_m = 0$ and $c \geq 0$. This implies that some column j satisfies $a_j = -\sum_{i\neq j} c_i' a_i$. where $c_i' \geq 0$.

Assume that A satisfies condition (1) and suppose that it satisfies condition (2). Then $p'a_i > 0 \quad i = 1, ..., N$ However, we know that $p'a_j = -\sum_{i \neq j} c_i' p' a_i$. Since $c_i \geq 0$ and $p'a_i > 0$, this term must be negative, which contradicts $p'a_i > 0$. Thus we cannot have both condition 1 and condition 2 holding.

Consider the set $S = \{Ax, x \geq 0\}$. Assume that $0 \notin S$. S is closed, nonempty and convex. By the separating hyper-plane theorem, there exists a vector p such that $p'0 < p'Ax$ where $x \geq 0$. Applying this multiple times, taking x to be each of the standard ordered basis, we arrive at $p'0 = 0 < p'A$. Therefore if condition (1) is not met, it must be the case that condition (2) is met.

This implies that we cannot have neither conditions holding, and since we proved above that we cannot have both conditions, the logical conclusion is that only one of the conditions may hold.

4 Part 2

4.1 1

The code for the revised simplex method is below:

```
function RevissedSimplexIteration( BInv::Matrix{Float64}, x::Vector{Float64},
                            A::Matrix{Float64}, c::Vector{Float64},
                            b::Vector{Float64}, k::Vector{Int64},
                            M::Int64, N::Int64)


    #First let us compute some costs, we stop computing costs as soon
    #as we have a negative cost
    j = -1
    #Numerical Precision problems when working with the inverse
    for i in 1:M
        if( c[i] - dot( c[k], BInv*A[:,i]) < -1e-8)
            j = i
            break
        end
    end

    #Check if we are at an optimal solution
    if( j == -1 )
        return 1
    end

    u = BInv*A[:,j]
    #Check if the problem is unbounded below
    if( sum(u[i] > 0 for i in 1:M ) == 0)
        return -1
    end


    #This is an implementation of Bland's Rule
    min = 1.0e10
    ℓ = -1
    for i in 1:M
        if( u[i] > 0.0 )
            thetaTemp = (x[k[i]] / u[i])
            #The strict inequality means that the first i wins the
            #tie.
            if( thetaTemp < min )
                min = thetaTemp
```

```
                ℓ = i
            end
        end
    end
    #Now the basis has k instead of ℓ
    k[ℓ] = j

    #Are elementary matrix operations faster than this?
    for i in 1:M
        if i != ℓ
            @inbounds BInv[i,:] -= (u[i] / u[ℓ])*BInv[ℓ,:]
        end
    end
    val = u[ℓ]
    for j in 1:M
        @inbounds BInv[ℓ,j] =  BInv[ℓ,j] / u[ℓ]
    end

    x .= 0
    x[k] = BInv*b

    #Continue iterating:
    return 0
end
```

For a simulation, since generating random matrices A, c, b is not very feasible, as there is no coded version of phase-II which obtains an initial basic feasible solution. Without a basic feasible solution, we wish to simulate problems in which there is a known basic feasible solution always, hopefully starting at zero.

To this end, I simulate the problem using different cases of quantile regression. In particular, I use $\tau = .5$ so that the Least-Absolute Deviations estimator can be used, as it is easier to simulate with thoughts towards a more robust calculation of the mean effect.

The true coefficients of the model are generated randomly, coming from the Uniform (-10,10), where there is no constant. I used five coefficients, and simulated data for different amounts of data. Run-times were averaged over many iterations in an attempt to capture run-time speed compared to speed costs related to allocation of memory that become relevant for even medium sized data sets.

Memory is important because there are M rows in the A matrix, and then there are $2M + 2p$ columns. This means that the storage requirements are $\mathcal{O}(M^2)$ and allocation becomes quite costly regardless of the speed of the algorithm. This could be ameliorated through better coding, but did not seem relevant for this assignment. The Table below summarizes the run times:

M	Simplex Run Time	Revised Run Time
250	0.300426	0.750987667
500	5.438479	10.307776333
750	25.473887	39.933402333
1000	21.178596667	28.797609333
1250	75.5935325	102.380060

It is important to note that the difficulty to converge of the algorithm is driven more by the shape, which was dictated by the randomly simulated data, both algorithms were run against the same data, so while both struggled with the draw at $M = 750$, the relative performance differences between the two can be looked at to draw conclusions.

One would expect the effects to be increasing in the model complexity, as the matrix B becomes more and more costly to invert, a $\mathcal{O}(N^3)$ operation. But this is not the case. For relative small values of M, the revised algorithm has a larger return, netting half the run time of the regular inversion technique. As the problem scales, this effect is still prominent, but begins to recede to only being three quarters of the run time of the regular algorithm.

Some of these differences have to be credited to the inversion techniques used in Julia, which exploit many of the structures of the matrix quite well, while my column operations are clumsy column-wise operations at best. It may be that generating elementary matrices and multiplying could exploit this structure better, but I was unable to find prefabricated elementary matrix code to test this hypothesis. It is clear that in my implementation, while there are performance gains from using the revised simplex method, there does not appear to be a reduction in the order of the complexity of the problem from maintaining the inverse B matrix via row operations.

4.2 2

Show equivalence of the two dual formulations of the quantile regression problem.

$$\begin{aligned} \max_{d} \quad & Y'd \\ \text{subject to:} \quad & X'd = 0 \\ & (\tau - 1)1_n \leq d \leq \tau 1_n \end{aligned}$$

and

$$\begin{aligned} \max_{a} \quad & Y'a \\ \text{subject to:} \quad & X'\alpha = (1-\tau)X'1_n \\ & \alpha \in [0,1]^n \end{aligned}$$

The second problem may be rewritten as:

$$\begin{aligned} \max_{a} \quad & Y'a \\ \text{subject to:} \quad & X'(\alpha - (1-\tau)1_n) = 0 \\ & \alpha \in [0,1]^n \end{aligned}$$

Allowing $d = \alpha - (1-\tau)1_n$ we see that the constraint that $\alpha \in [0,1]^n$ is the same as $d \geq -(1-\tau)1_n$ and $d \leq 1 - (1-\tau)1_n$ This can be summarized as:

$$(\tau - 1)1_n \leq d \leq \tau 1_n$$

Therefore the optimization question becomes:

$$\begin{aligned} \max_{d} \quad & Y'd \\ \text{subject to: } & X'd = 0 \\ & (\tau - 1)1_n \leq d \leq \tau 1_n \end{aligned}$$

This is exactly the initial formulation, so the problems must be equivalent.

4.3 3

A box constrained problem can always be rewritten in a dimension of $2n$ where there is both an x and a slack variable s such that $x + s = 1$ and $x, s \geq 0$. By nature of this constraint, and the fact that we are iterating across basic feasible solutions, either x_i or s_i will be in the basis, with the possibility of both.

In problem (2) there are $n + m$ equality constraints, and $2n$ parameters, as we need a slack for each parameter. There will then be $n - m$ parameters in the active basis at any time.

Let us begin with some basic feasible solution, and a working version of B^{-1}. Compute $\bar{c}_j = c_j - c_B' B^{-1} A_j$. If $\bar{c}_j \geq 0$ then we have an optimal solution, if not, choose some j such that $\bar{c}_j < 0$

Now, we wish to have A_j enter the basis. Since it impossible for there to be neither x_i or s_i in the basis, the only elements that need to be considered for removal from the basis are the terms with x_i, s_i both active, or the $j + n \mod 2n$ term.

To this end we compute $u = B^{-1} A_j$ as before. However in the computation of θ, we need only consider the elements above.

$$\theta^* = \min_{i, u_i > 0 \cap \{i, i+n \mod 2n \in B \,\cup\, i = n+j \mod 2n\}} \frac{x_{B(i)}}{u_i}$$

We then replace the basis as before, letting ℓ fulfill the above minimization problem, and computing the values of B^{-1} using the same elementary row operations.

In psuedo-code form, this algorithm means following the instructions:

Assuming that we begin with some basic feasible solution.

1. Compute $\bar{c}_j$ as above, if it is all non-negative, then the solution is optimal.
2. Choose some j such that $\bar{c}_j < 0$.
3. Compute $u - B^{-1} A_j$
4. For each element of u:
   (a) If $u_i \leq 0$, skip this element.
   (b) If $i + n \mod 2n$ is not in the basis B, and $i \neq n + j \mod 2n$, skip this element.
   (c) Find the argmin of $\frac{x_{B(i)}}{u_i}$, call this index ℓ.

5. Exchange ℓ for j in the basis, updating the inverse matrix using the appropriate elementary matrix operations.

6. Compute $x_B = B^{-1}b$, and set $x_{-B} = 0$.

4.4 4

The code for the Barrodale and Roberts algorithm is given below.

```
function Barrodale( X::Matrix{Float64}, x::Vector{Float64},
                    k::Vector{Int64}, b::Vector{Float64},
                    c::Vector{Float64}, BInv::Matrix{Float64},
                    p::Int64, M::Int64, N::Int64)

    cBar = Vector{Float64}(undef,2*p)

    ℓ = 0
    u = Vector{Float64}(undef,0)

    for i in 1:p
        j = -1
        min = 1e10
        #Which β should enter the distribution?
        for z in 1:p
            cBar[z] = c[z] - dot( c[k], BInv*X[:,z])
            if cBar[z] < min
                j = z
                min = cBar[z]
            end
        end
        for z in (p+1):2*p
            cBar[z] = c[z] - dot( c[k], -BInv*X[:,z-p])
            if cBar[z] < min
                j = z
                min = cBar[z]
            end
        end
        #j is now the smallest element of cBar

        if j <= p
            ℓ,u = ChangeSignPivots( c, BInv, X[:,j], x, k, b, p, M, j)
        else
            ℓ,u = ChangeSignPivots( c, BInv, -X[:,j-p], x, k, b, p, M, j)
        end

        # Now we do a normal pivot bringing in β[j]
        k[ℓ] = j

        #Are elementary matrix operations faster than this?
        for z in 1:M
            if( z == ℓ)
                continue
            end
            BInv[z,:] -= (u[z] / u[ℓ])*BInv[ℓ,:]
        end
        BInv[ℓ,:] ./=  u[ℓ]

        #Compute the new x value
        x .= 0.0
        x[k] = BInv*b
    end
    #Phase 1 complete.
    println( x[1:2*p])

```

```
    cBar = Vector{Float64}(undef,2*M)
    min = 1e10
    j = -1
    for i in 1:M#(2*p+1):(2*p+M)
        cBar[i] = c[i+2*p] - dot( c[k], BInv[:,i])
        if cBar[i] < min
            j = i+2*p
            min = cBar[i]
        end
    end
    for i in 1:M#Note that the second half of the residuals uses -I
        cBar[M+i] = c[M+i+2*p] - dot( c[k], -BInv[:,i])
        if cBar[i] < min
            j = i+2*p+M
            min = cBar[M+i]
        end
    end

    # We stop once all reduced costs are positive.
    while( min < 0.0)
        #Since we know we are using A[:,j] where j is a standard
        #ordered basis, we just need to make sure we get the element
        #correct. Lots of silly modular shit to do that
        sob = zeros(M)
        sob[((j-2*p-1)%M)+1] = 1.0-2.0(j-2*p > 2*p+M)
        ℓ,u = ChangeSignPivots( c, BInv, sob, x, k, b, p, M, j)
        # Now we do a normal pivot bringing in β[j]
        k[ℓ] = j

        #Are elementary matrix operations faster than this?
        for z in 1:M
            if( z == ℓ)
                continue
            end
            BInv[z,:] -= (u[z] / u[ℓ])*BInv[ℓ,:]
        end
        BInv[ℓ,:] ./=  u[ℓ]

        #Recalculate all of the reduced costs for u,v
        min = 1e10
        j = -1
        for i in 1:M#(2*p+1):(2*p+M)
            cBar[i] = c[i+2*p] - dot( c[k], BInv[:,i])
            if cBar[i] < min
                j = i+2*p
                min = cBar[i]
            end
        end
        for i in 1:M#(2*p+1):(2*p+M)
            cBar[M+i] = c[M+i+2*p] - dot( c[k], -BInv[:,i])
            if cBar[i] < min
                j = i+2*p+M
                min = cBar[M+i]
            end
        end
    end

    #Compute the new x value
    x .= 0.0
    x[k] = BInv*b
    return x
end

function ChangeSignPivots( c::Vector{Float64}, BInv::Matrix{Float64},
                           Aj::Vector{Float64}, x::Vector{Float64},
                           k::Vector{Int64}, b::Vector{Float64},
                           p::Int64, M::Int64, j::Int64)
```

```
    exiting = 0
    ℓ = -1

    u = Vector{Float64}

    while( c[j] - dot( c[k], BInv*Aj) < 1e-8)
        u = BInv*Aj
        min = 1.0e10
        ℓ = -1
        for z in 1:M
            if( u[z] > 0.0 )
                thetaTemp = (x[k[z]] / u[z])
                #The strict inequality means that the first i wins the
                #tie.
                if( thetaTemp < min )
                    min = thetaTemp
                    ℓ = z
                end
            end
        end

        if ℓ == -1
            println("ℓ issue")
            return -1, u
        end

        #Now we know we want k[ℓ] to exit, we need to find its
        #negative (positve) counter-part
        exiting = k[ℓ]
        # If we need to do a beta change of sign
        if exiting <= 2*p
            # Note that Julia is 1-indexed so for modular
            # arthimetic we need to minus one, mod, then add one
            entering = (exiting + p-1) % (2*p)+1
        else
            entering = ((exiting-2*p)+M-1)%(2*M) + 2*p+1
        end

        k[ℓ] = entering
        BInv[ℓ,:] *= -1
        x .= 0.0
        x[k] = BInv*b
    end
    #If we had a negative Reduced cost before, we should undo the
    #last change, and do a regular pivot. These pivots are cheap,
    #so doing 2 more pivots than we need isn't a big deal
    k[ℓ] = exiting
    BInv[ℓ,:] *= -1

    return  ℓ,u
end
```

In order to test the performance of my simulation against the Gurobi implementation, I use the same simulation specifics as in question 1. This allows for an easy comparison between my naive-simplex approach, the Barrodale-Roberts coded algorithm, and the Gurobi “big boy” software.

In order to access Gurobi, I use Julia for Mathematical Programming, which functions as a solver interface much like AMPL. The code used to answer the problem is given below:

```
function GurobiQuantReg( X::Matrix{Float64},
                         Y::Vector{Float64},
                         τ::Float64,
                         p::Int64, M::Int64, N::Int64)
```

```


    m = Model(solver = GurobiSolver())
    @variable( m, β□[1:p] >= 0)
    @variable( m, β□[1:p] >= 0)
    @variable( m, u[1:M] >= 0)
    @variable( m, v[1:M] >= 0)
    @constraint( m, fit[i=1:M],
                 sum( X[i,j]*β□[j] for j in 1:p )
                 - sum( X[i,j]*β□[j] for j in 1:p)
                 + u[i] - v[i] == Y[i] )
    @objective( m, Min, τ*sum( u[i] for i in 1:M )
                + (1-τ)*sum(v[i] for i in 1:M) )
    status = solve(m)
    println(getvalue(β□))
    println(getvalue(β□))
    return [getvalue(β□), getvalue(β□), getvalue(u), getvalue(v)]
end
```

The results of the horse-race are summarized below. The professional solver is orders of magnitude faster than even the Barrodale-Roberts algorithm. It is worth noting that the results from Gurobi are computed from using a seperate machine which was considerably slower than the others, so this table understates the advantages of using Gurobi.

M	Revised Simplex	Regular Simplex	Barrodale Roberts	Gurobi
250	0.300426	0.750987667	0.292047333	0.034217
500	5.438479	10.307776333	0.473022	0.065429
750	25.473887	39.933402333	2.173836	0.078337
1000	21.178596667	28.797609333	2.868348	0.156235
1250	75.5935325	102.380060	19.424499667	0.170933

The same mechanics of using the same random seed and data generation are held constant across the four methods. Much of the cost of these methods can be attributed to the construction of the A matrix. The Barrodale-Roberts and Gurobi methods do not require that this be constructed and stored in memory, so there is a large gain in terms of allocation and garbage collection. However it is clear that there is more to this than simply that, as we see enormous differences between the methods.

For the Gurobi solver, the number of iterations was always 1-5 iterations more than the number of data points. It is difficult to contrast the Barrodale-Roberts algorithm to this, as it made many inefficient but cheap iterations in the form of change of sign pivots, so counting iterations may not be a clear way of measuring its effectiveness. However as the computation time suggests, it is still significantly slower than the professional solver that is able to further exploit the structure of the model. It is clear that there are enormous gains from using commercial software, though the Barrodale-Roberts algorithm performs extremely well when compared against a traditional simplex algorithm, it cannot attempt to compete with serious proprietary software maintained by professionals.