Tony Tohme - 18.338 Project

In this notebook, we review and implement the simplex method, an algorithm that solves Linear Programming (LP) problems. We also perform some experiments with random matrices to find interesting observations and conjectures.

The LP problem in standard form is expressed as:

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

where $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{b}$ is an $m \times 1$ vector, where m is the number of equality constraints and n is the number of decision variables (including the slack variables). It is worth noting that maximizing $\mathbf{c}'\mathbf{x}$ is simply equivalent to minimizing $-\mathbf{c}'\mathbf{x}$.

Let $P = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ be a polyhedron corresponding to the feasible set of the LP problem in standard form. We also assume that the m rows of $\mathbf{A}$ are linearly independent. Then, an optimal solution to the LP problem (if it exists) tends to occur at a ``corner'' of P (known as vertex, or basic solution).

If an LP problem in standard form has an optimal solution, then there exists a basic feasible solution (a vertex of P) that is optimal. The simplex algorithm is based on this fact and searches for an optimal solution by moving from one basic feasible solution to another, along the edges of the feasible set P, always in a cost reducing direction. The algorithm terminates when we reach a basic feasible solution (that is optimal) at which none of the available edges leads to a cost reduction.

An iteration of the simplex algorithm

• We start with a basis consisting of the basic columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(m)}$, and an associated basic feasible solution $\mathbf{x}$.

• We compute the reduced costs $\bar{c}_j = c_j - \mathbf{c}'_B\mathbf{B}^{-1}\mathbf{A}_j$ for all nonbasic indices j. If they are all nonnegative, the current basic feasible solution is optimal, and the algorithm terminates; else, we choose some j for which $\bar{c}_j < 0$.

• We then compute $\mathbf{u} = \mathbf{B}^{-1}\mathbf{A}_j$. If no component of $\mathbf{u}$ is positive, the optimal cost is $-\infty$, and the algorithm terminates (the problem is thus unbounded).

• If some component of $\mathbf{u}$ is positive, we let $\theta^* = \min_{\{i=1,\ldots,m | u_i > 0\}} \frac{x_{B(i)}}{u_i}$.

• We let ℓ be such that $\theta^* = x_{B(\ell)}/u_\ell$. We form a new basis by replacing $\mathbf{A}_{B(\ell)}$ with $\mathbf{A}_j$. If $\mathbf{y}$ is the new basic feasible solution, the values of the new basic variables are $y_j = \theta^*$ and $y_{B(i)} = x_{B(i)} - \theta^* u_i$, $i \neq \ell$.

An iteration of the full tableau implementation

• We start with the tableau associated with a basis matrix $\mathbf{B}$ and the corresponding basic feasible solution $\mathbf{x}$.

• We examine the reduced costs in the zeroth row (the top row) of the tableau. If they are all nonnegative, the current basic feasible solution is optimal, and the algorithm terminates; else, we choose some j for which $\bar{c}_j < 0$.

• We then consider the vector $\mathbf{u} = \mathbf{B}^{-1}\mathbf{A}_j$, which is the jth column (the pivot column) of the tableau. If no component of $\mathbf{u}$ is positive, the optimal cost is $-\infty$, and the algorithm terminates.

• For each i for which u_i is positive, we compute the ratio $x_{B(i)}/u_i$. We let ℓ be the index of a row that corresponds to the smallest ratio. The column $\mathbf{A}_{B(\ell)}$ exits the basis and the column $\mathbf{A}_j$ enters the basis.

• We add to each row of the tableau a constant multiple of the ℓth row (the pivot row) so that u_ℓ (the pivot element) becomes one and all the other entries of the pivot column become zero.

We now introduce the concept of degeneracy. A basic solution $\mathbf{x} \in \mathbb{R}^n$ is said to be degenerate if more than n constraints are active (i.e. binding) at $\mathbf{x}$. If the problem is in its standard form, then $\mathbf{x}$ is degenerate if more than $n - m$ components of $\mathbf{x}$ are zero, i.e. if some components of $\mathbf{x}_B$ are zero. The concept of degeneracy is crucial in the development of the simplex algorithm since it may lead to a scenario known as cycling (see below).

Note that in the presence of degeneracy, the algorithm might change the basis but stay at the same basic feasible solution. In this case, there are two possible scenarios. On the one hand, a sequence of such basis changes may lead to the discovery of a cost reducing feasible direction. On the other hand, a sequence of basis changes might lead back to the initial basis and in this case the algorithm may loop indefinitely. This second scenario is known as $\textit{cycling}$.

To avoid cycling, we will follow the $\textit{smallest subscript}$ rule (known as Bland's rule) for both the entering and the exiting variables.

Smallest subscript pivoting rule

• We find and choose the smallest j for which the reduced cost $\bar{c}_j$ is negative and have the column $\mathbf{A}_j$ enter the basis.

• Out of all variables x_i that are tied in the test for choosing an exiting variable, we select the one with the smallest value of i.

By adopting this pivoting rule, cycling is avoided and the simplex algorithm is guaranteed to terminate after a finite number of iterations.

The big-M method

From the two algorithms presented above, note that we start the simplex algorithm by finding an initial basic feasible solution. However, this is not always straightforward, and in general, finding an initial basic feasible solution requires us to add some *artificial variables* and solve the problem in a different way. In this project, we adopt an approach known as the *big-M method*. Consider the problem

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

Note that we can assume, without loss of generality, that $\mathbf{b} \geq \mathbf{0}$ (by multiplying some of the equality constraints by -1). We introduce a vector $\mathbf{y} \in \mathbb{R}^m$ of artifical variables and we modify the cost function to solve the following LP problem

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} + M\sum_{i=1}^{m} y_i \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{y} = \mathbf{b} \\ & \mathbf{x} \geq 0 \\ & \mathbf{y} \geq 0 \end{aligned}$$

where M is a large positive constant. By following this approach, initialization of the simplex algorithm becomes easy: we let $\mathbf{x} = \mathbf{0}$ and $\mathbf{y} = \mathbf{b}$, and we get a basic feasible solution with corresponding basis matrix being the identity.

Note that for large M, if the original problem is feasible and its optimal cost is finite (i.e. if the problem is not unbounded), all of the artificial variables y_i will be 0 (i.e. $\mathbf{y} = \mathbf{0}$) and this will lead to the minimization of the original cost function $\mathbf{c}'\mathbf{x}$. Therefore, if the optimal solution of the problem above contains a nonzero artificial variable, we can conclude that the original problem is infeasible.

In [18]:
```
using LinearAlgebra

function MySimplex(A, b, c, cstr_types, LP_type)

#=  Description:

        This function represents my implementation of the simplex method for LP
        problems, as explained in Chapter 3 of the book "Introduction to Linear
        Optimization" by Dimitris Bertsimas and John N. Tsitsiklis. I will adopt
        the full tableau implementation and follow the big-M method, using the
        smallest subscript pivoting rule.

    Inputs Arguments:

        A - a matrix of coefficients of the left-hand side of the constraints.
        b - a column vector of the right-hand side constant of the constraints.
        c - a cost (column) vector containing the coefficients of the decision variables in the cost function.
        cstr_types - a column vector indicating the type of each constraint; 1 for >=, 0 for =, and -1 for <= constrai
nts.
        LP_type - a parameter indicating the type of optimization; 1 for minimization and 0 for maximization.

    Outputs:

        optimal_sol  - a vector of the optimal solution.
        optimal_cost - a scalar indicating the optimal cost.
        iters        - a scalar indicating the number of iterations. =#

    M = 1e+8 # choose an appropriately large M

    m, n = size(A)

    if length(c) != n || length(b) != m
        @assert(false, "Dimension mismatch!")
    end

    # convert maximization LP problem to a minimization LP problem
    if LP_type == 0
       c = -c
    end

    # make the b vector non-negative
    for i = 1:length(b)
        if b[i] < 0
            A[i,:] = -A[i,:]
            b[i] = -b[i]
            cstr_types[i] = -cstr_types[i]
        end
    end

    B_indices = []

    # add slack variables to convert the problem to standard form
    slack_nb = 0

    for i = 1:length(cstr_types)
        if cstr_types[i] == 1
            cstr_types[i] = 0
            slack_nb += 1
            A = [A zeros(m,1)]
            A[i, end] = -1
            c = [c; 0]
        elseif cstr_types[i] == -1
            slack_nb += 1
            A = [A zeros(m,1)]
            A[i, end] = 1
            c = [c; 0]
            push!(B_indices, n+slack_nb)
        elseif cstr_types[i] != 0
            @assert(false, "Wrong constraint type!")
        end
    end

    # introduce artificial variables and modify the cost function
    art_nb = 0

    for i = 1:length(cstr_types)
        if cstr_types[i] == 0
            Iₘ = 1 * Matrix(1.0I, m, m)
            if any(sum(A .== Iₘ[:,i], dims = 1) .== m)
                B_indices = push!(B_indices, findall(sum(A .== Iₘ[:,i], dims = 1) .== m)[1][2])
            else
                art_nb += 1;
                A = [A zeros(m,1)]
                A[i, end] = 1
                c = [c; M]
                push!(B_indices, n+slack_nb+art_nb)
            end
        end
    end

    # compute the first basic feasible solution
    B = A[:,B_indices]
    x_B = B\b
    c_B = c[B_indices]

    optimal_cost = - c_B'*(B\b)
    reduced_cost = c' - c_B'*(B\A)
    optimal_sol = zeros(n+slack_nb+art_nb,1)
    u = B\A

    # display the tableau
#     println([optimal_cost, reduced_cost])
#     for i=1:size(x_B,1)
#         println(x_B[i,:],u[i,:])
#     end
#     println(B_indices,'\n')

    iters = 0
    while !all(reduced_cost.>=0)
        iters += 1
        pivot_col = findall(reduced_cost .< 0)[1][2]

        # check if the LP problem is unbounded
        if all(u[:,pivot_col].<=0)
            optimal_sol = []
            optimal_cost = - Inf
            if LP_type == 0
                optimal_cost = -optimal_cost
            end
            print("The given LP problem is unbounded!")
            return optimal_sol, optimal_cost, iters
        else
            ratio = Inf*ones(m,1)
            ratio[u[:,pivot_col].>0] = x_B[u[:,pivot_col].>0]./u[u[:,pivot_col].>0,pivot_col]
            pivot_row = argmin(ratio)[1]
            x_B[pivot_row] = x_B[pivot_row]./u[pivot_row, pivot_col]
            B_indices[pivot_row] = pivot_col
            u[pivot_row,:] = u[pivot_row,:]./u[pivot_row, pivot_col]
            for i = 1:m
                if i != pivot_row
                    x_B[i] = x_B[i] - u[i, pivot_col]*x_B[pivot_row]
                    u[i,:] = u[i,:] .- u[i, pivot_col].*u[pivot_row,:]
                end
            end
            optimal_cost = optimal_cost .- reduced_cost[pivot_col]*x_B[pivot_row]
            reduced_cost = reduced_cost .- (reduced_cost[pivot_col].*u[pivot_row,:])'

            # display the tableau
#             println([optimal_cost, reduced_cost])
#             for i=1:size(x_B,1)
#                 println(x_B[i,:],u[i,:])
#             end
#             println(B_indices,'\n')

        end
    end

    optimal_sol[B_indices] = x_B

    # check if the LP problem is infeasible
    if !all(B_indices .<= n+slack_nb)
        if !all(optimal_sol[B_indices[B_indices .> n+slack_nb]] .== 0)
            optimal_sol = []
            optimal_cost = []
            print("The given LP problem is infeasible!")
            return optimal_sol, optimal_cost, iters
        end
    end

    # return the optimal solution and cost
    optimal_sol = optimal_sol[1:n]
    optimal_cost = - optimal_cost

    if LP_type == 0
        optimal_cost = - optimal_cost
    end

    return optimal_sol, optimal_cost, iters
end
```

```
Out[18]: MySimplex (generic function with 1 method)
```

Bounded and Feasible LP

$$\begin{array}{ll} \text{minimize} & 5x_1 + 2x_2 - 4x_3 \\ \text{subject to} & 6x_1 + \ \ x_2 - 2x_3 \geq 5 \\ & \ \ x_1 + \ \ x_2 + \ \ x_3 \leq 4 \\ & 6x_1 + 4x_2 - 2x_3 \geq 10 \\ & \qquad x_1, x_2, x_3 \geq 0 \end{array}$$

The corresponding inputs to our function `MySimplex` are as follows:

$$\mathbf{A} = \begin{bmatrix} 6 & 1 & -2 \\ 1 & 1 & 1 \\ 6 & 4 & -2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 5 \\ 2 \\ -4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 4 \\ 10 \end{bmatrix}, \quad \mathbf{cstr\_types} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}, \quad \mathbf{LP\_type} = 1.$$

The true optimal cost for this LP problem is 3.

In [2]:
```
A = [[6,1,6]  [1,1,4]  [-2,1,-2]]; b = [5,4,10]; c = [5,2,-4]; cstr_types = [1,-1,1]; LP_type = 1;
```

In [3]:
```
optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type)
println("x₁ = ", optimal_sol[1], ", x₂ = ", optimal_sol[2], ", x₃ = ", optimal_sol[3])
println("optimal cost = ", optimal_cost)
println("# iterations = ", iters)
```

```
x₁ = 1.0, x₂ = 1.6666666666666667, x₃ = 1.3333333333333333
optimal cost = 2.9999999006589255
# iterations = 3
```

In [4]:
```
# @code_warntype MySimplex(A, b, c, cstr_types, LP_type)
```

Unbounded LP

$$\begin{array}{ll} \text{minimize} & -10x_1 - 12x_2 - 12x_3 \\ \text{subject to} & \ \ x_1 + \ 2x_2 + \ 2x_3 \geq 20 \\ & 2x_1 + \ \ x_2 + \ 2x_3 \geq 20 \\ & 2x_1 + \ 2x_2 + \ \ x_3 \geq 20 \\ & \qquad x_1, x_2, x_3 \geq 0 \end{array}$$

The corresponding inputs to our function `MySimplex` are as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -10 \\ -12 \\ -12 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 20 \\ 20 \\ 20 \end{bmatrix}, \quad \mathbf{cstr\_types} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{LP\_type} = 1.$$

In [5]:
```
A = [[1,2,2]  [2,1,2]  [2,2,1]]; b = [20,20,20]; c = [-10,-12,-12]; cstr_types = [1,1,1]; LP_type = 1;
```

In [6]:
```
optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
```

```
The given LP problem is unbounded!
```

Infeasible LP

$$\begin{array}{ll} \text{minimize} & x_1 + x_2 \\ \text{subject to} & x_1 \qquad \geq 6 \\ & \qquad x_2 \geq 6 \\ & x_1 + \ x_2 \leq 11 \\ & \ \ x_1, x_2 \geq 0 \end{array}$$

The corresponding inputs to our function `MySimplex` are as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 11 \end{bmatrix}, \quad \mathbf{cstr\_types} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, \quad \mathbf{LP\_type} = 1.$$

In [7]:
```
A = [[1,0,1]  [0,1,1]]; b = [6,6,11]; c = [1,1]; cstr_types = [1,1,-1]; LP_type = 1;
```

In [8]:
```
optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
```

```
The given LP problem is infeasible!
```

LP Test Problems

Choose from **agg2.mat**, **agg3.mat**, **israel.mat**, **lotfi.mat**, **share1b.mat**.

In [9]:
```
using MAT
vars = matread("agg2.mat")
A, b, c, cstr_types, LP_type, true_optimal_cost = vars["A"], vars["b"], vars["c"], vars["cstr_types"], vars["LP_type"
], vars["true_optimal_cost"];
```

In [10]:
```
optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
println("true optimal cost = ", true_optimal_cost)
println("optimal cost = ", optimal_cost[1])
println("# iterations = ", iters)
```

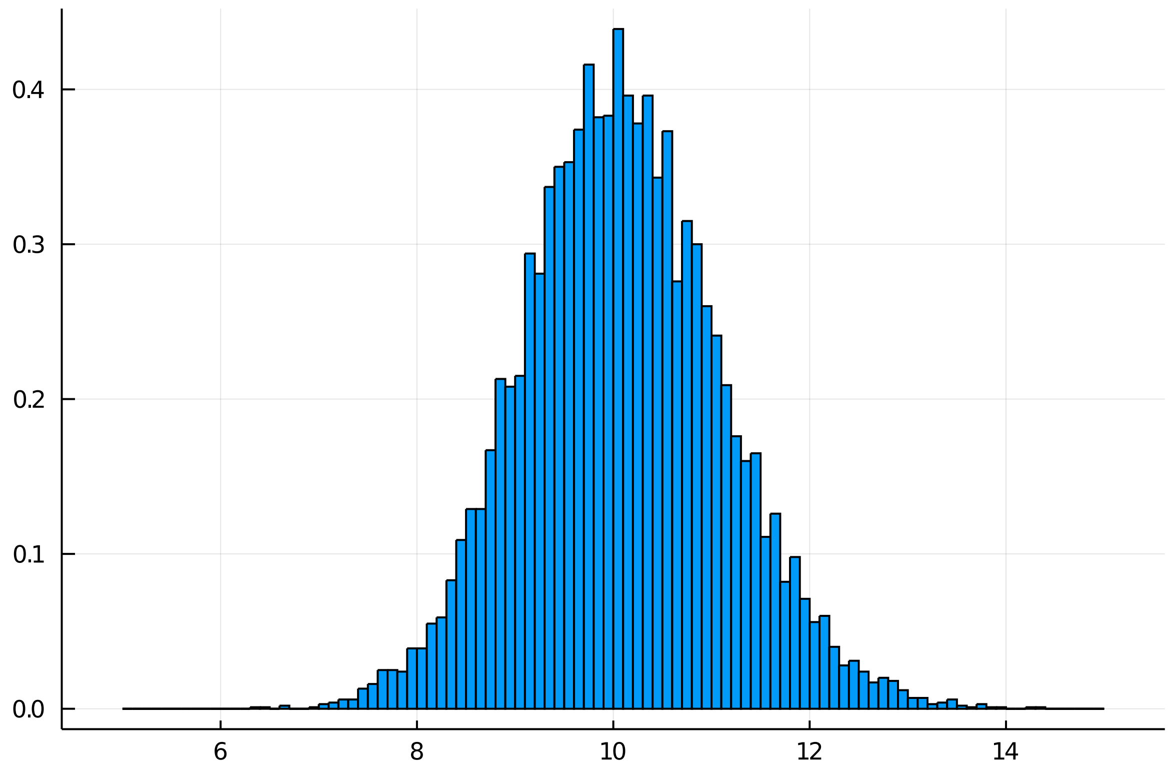
```
true optimal cost = -2.0239252356e7
optimal cost = -2.02392523466796 88e7
# iterations = 229
```

Random Matrices

In [12]:
```
using Plots, Random, Distributions
```

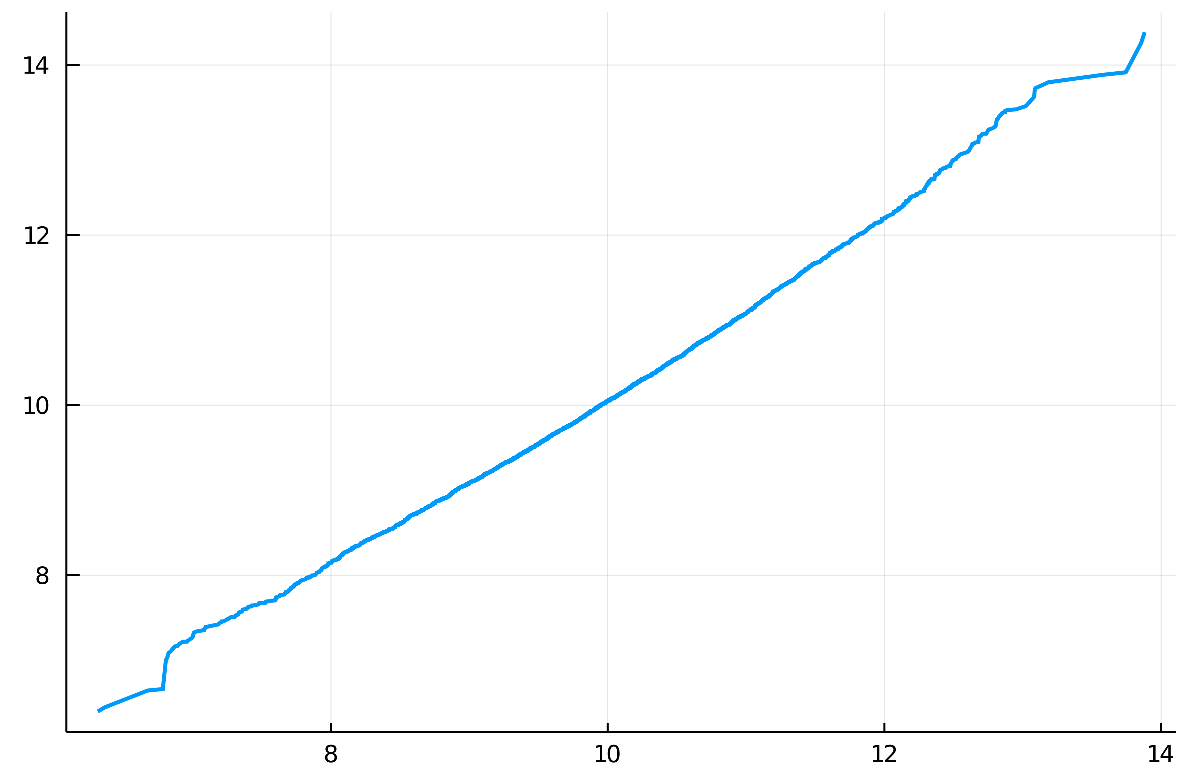
In [13]:
```
n = 3
n_samples = 10^4
X = zeros(n_samples,n)
opt_cost = zeros(n_samples,1)
index = 0
μ = 10
while index < n_samples
    A = rand(Normal(μ),n,n)
    b = rand(Normal(μ),n)
    c = rand(Normal(μ),n)
    cstr_types = zeros(n,1)
    LP_type = 1
    optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type)
    if !isempty(optimal_sol)
        index += 1
        X[index,:] = optimal_sol
        opt_cost[index] = optimal_cost
    end
end

Plots.histogram(opt_cost, normalize=true, bins= 5:.1:15, bar_width = 0.1, legend = false)
```

Out[13]:



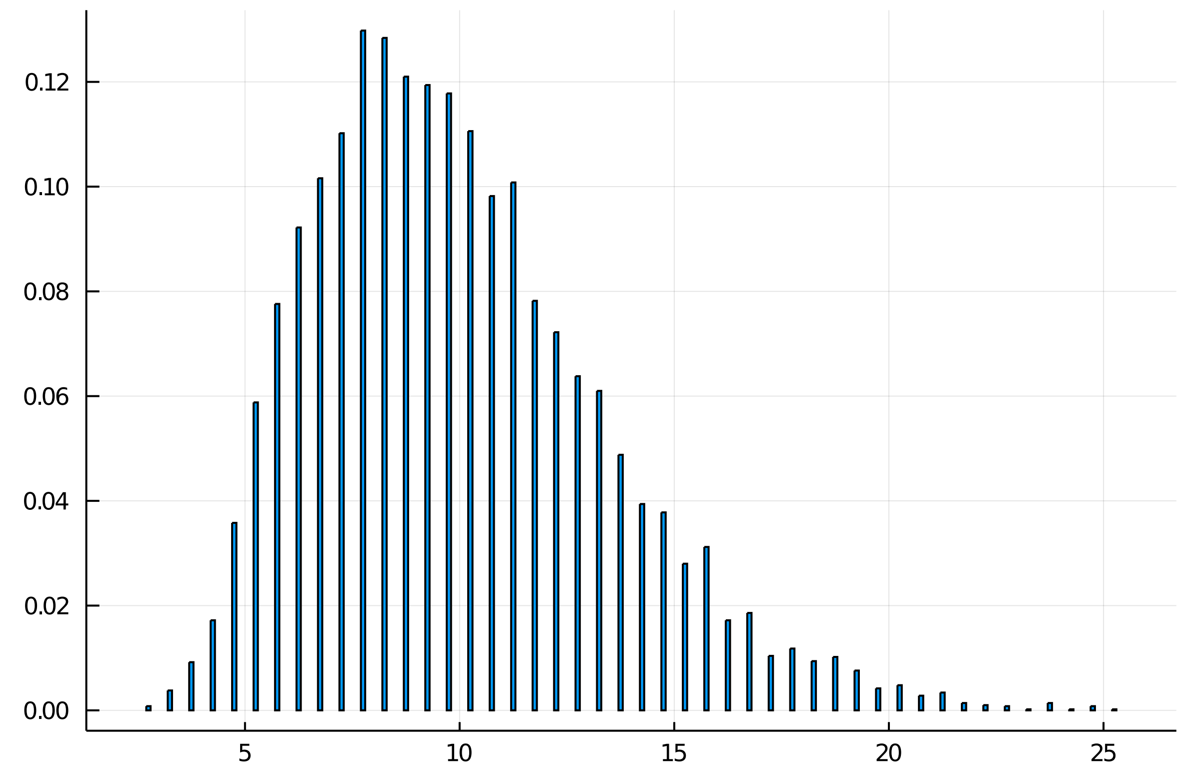
In [14]:
```
plot(sort(rand(Normal(μ),n_samples)), sort(opt_cost[:]), linewidth = 2, legend = false)
```

Out[14]:



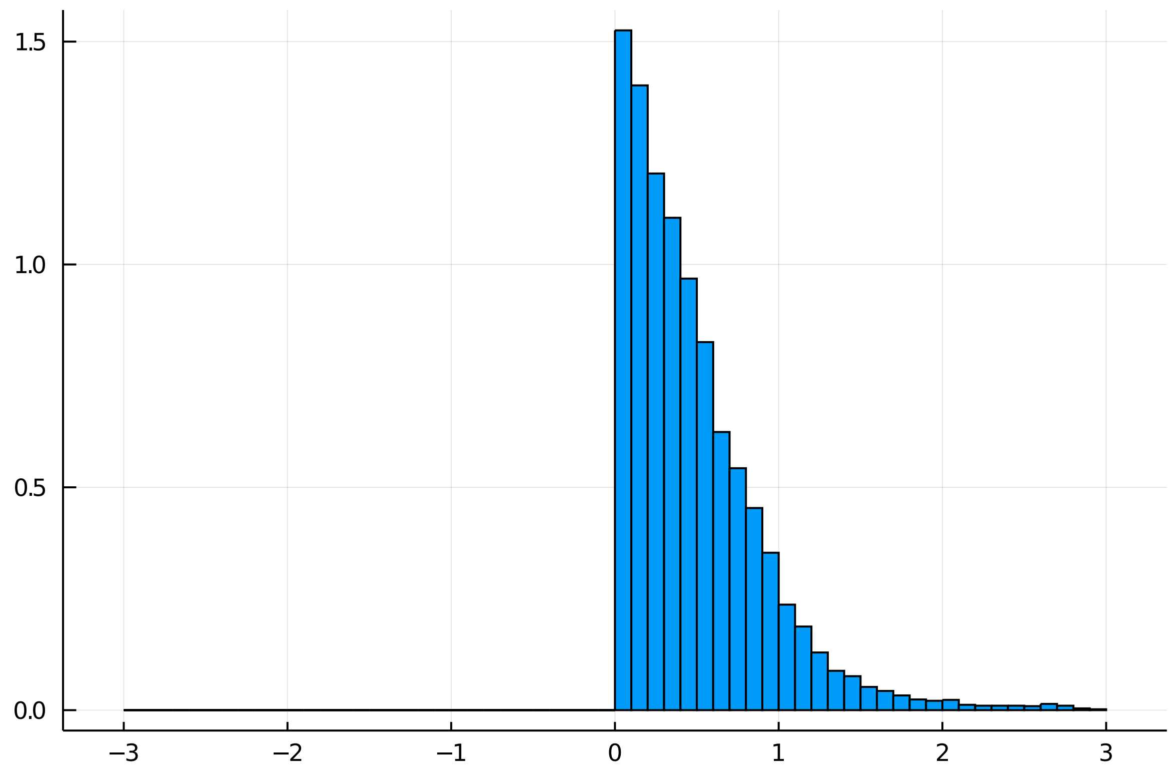
In [15]:
```
n = 3
n_samples = 10^4
X = zeros(n_samples,n)
opt_cost = zeros(n_samples,1)
index = 0
lower = 5
upper = 15
while index < n_samples
    A = rand(Uniform(lower,upper),n,n)
    b = rand(Uniform(lower,upper),n)
    c = rand(Uniform(lower,upper),n)
    cstr_types = ones(n,1)
    LP_type = 1
    optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type)
    if !isempty(optimal_sol)
        index += 1
        X[index,:] = optimal_sol
        opt_cost[index] = optimal_cost
    end
end

Plots.histogram(opt_cost, normalize=true, bar_width = 0.1, legend = false)
```

Out[15]:



In [16]:
```
n = 3
n_samples = 10^4
X = zeros(n_samples,n)
opt_cost = zeros(n_samples,1)
index = 0
while index < n_samples
    A = rand(n,n)
    b = rand(n)
    c = rand(n)
    cstr_types = ones(n,1)
    LP_type = 1
    optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type)
    if !isempty(optimal_sol)
        index += 1
        X[index,:] = optimal_sol
        opt_cost[index] = optimal_cost
    end
end

Plots.histogram(opt_cost, normalize=true, bins= -3:.1:3, bar_width = 0.1, legend = false)
```

Out[16]:



Comparison with the discourse thread

https://discourse.julialang.org/t/is-it-time-for-a-native-julia-lp-simplex-solver/21639 (https://discourse.julialang.org/t/is-it-time-for-a-native-julia-lp-simplex-solver/21639)

Code taken from: https://github.com/IainNZ/RationalSimplex.jl/blob/master/src/RationalSimplex.jl (https://github.com/IainNZ/RationalSimplex.jl/blob/master/src/RationalSimplex.jl)

In [17]:
```julia
# module RationalSimplex

# import LinearAlgebra: I

# export simplex

"""
    simplex(c, obj, A, b, senses)
Solve the linear program
     {min/max}  dot(c, x)
    subject to  A x {>=|==|<=} b
                x >= 0,
where `x` is in the set of Rationals (so algorithm is exact).
`obj` should be either `:Min` or `:Max`.
`senses` should be a `Vector` of `Char`s: `'<','=','>'`.
"""
function simplex1(c::Vector{T}, obj::Symbol, A::Matrix{T}, b::Vector{T},
                 senses::Vector{Char}) where {T <: Rational}
    # Any constraint that isn't an equality needs an auxiliary variable.
    num_auxiliary = count(sense -> sense != '=', senses)
    num_constraints, num_variables = size(A)
    num_variables_aux = num_variables + num_auxiliary
    # Extend constraint matrix and objective function, normalize objective.
    A_aux = zeros(T, num_constraints, num_variables_aux)
    A_aux[:, 1:num_variables] = A
    c_aux = zeros(T, num_variables_aux)
    c_aux[1:num_variables] = (obj == :Max) ? -c : c
    b_aux = copy(b)  # In case we modify signs.
    # Add the auxiliaries to the constraint matrix, and make sure b ≥ 0.
    offset = 1
    for (row, sense) in enumerate(senses)
        if sense == '<'
            A_aux[row, num_variables + offset] = one(T)
            offset += 1
        elseif sense == '>'
            A_aux[row, num_variables + offset] = -one(T)
            offset += 1
        end
        if b[row] < zero(T)
            A_aux[row, :] *= -one(T)
            b_aux[row] *= -one(T)
        end
    end
    # Solve problem with auxiliaries, but remove auxiliaries before returning.
    sense, x = simplex(c_aux, A_aux, b_aux)
    return sense, x[1:num_variables]
end


"""
    simplex(c, A, b)
Solve the linear program in standard computational form
           min  dot(c, x)
    subject to  A x == b [≥ 0]
                x >= 0,
where `x` is in the set of Rationals (so algorithm is exact).
The algorithm is the "two-phase primal revised simplex method".
In the first phase auxiliaries are created to get an initial basis.
We then eliminate the auxiliaries from the basis until it consists only of
actual variables. This is the "textbook" algorithm, and shouldn't be used for
anything that matters. It doesn't exploit sparsity at all. You could use it
with floating points but it won't work for anything except the most simple
problem due to accumulated errors and comparisons with zero.
"""
function simplex(c::Vector{T}, A::Matrix{T}, b::Vector{T}) where {T<:Rational}
    @assert all(b .> zero(T))

    # Set up data structures.
    num_constraints, num_variables = size(A)
    is_basic = zeros(Bool, num_variables + num_constraints)
    basic = zeros(Int, num_constraints)  # Indices of current basis.
    Binv = Matrix{T}(I, num_constraints, num_constraints)  # Basis inverse.
    cB = ones(T, num_constraints)  # Costs of basic variables.
    x = zeros(T, num_variables + num_constraints)  # Current solution.

    # Intialize phase one of RSM by setting basis = auxiliaries.
    for aux_index in 1:num_constraints
        basic[aux_index] = num_variables + aux_index
        is_basic[num_variables + aux_index] = true
        x[num_variables + aux_index] = b[aux_index]
    end
    phase_one = true

    # Begin simplex iterations.
    status = :Unknown
    while true
        # Calculate dual solution.
        π = vec(cB' * Binv)
        # Use it to calculate the reduced costs of the variables. Don't
        # calculate for auxiliaries - they can't re-enter the basis.
        rc = (phase_one ? zero(T) : c) .- vec(π' * A)
        @. rc *= !is_basic[1:num_variables]  # In basis, so don't consider.
        # Find variable to enter basis.
        min_rc, entering = findmin(rc)
        # If none have negative reduced cost, there are no variables that can
        # enter, and we are at optimality for this phase. This may or may not
        # be optimal for the actual problem.
        if min_rc >= zero(T)
            if phase_one
                phase_one = false
                # If any auxiliary still nonzero, we couldn't find a feasible
                # basis without auxiliaries.
                if any(x[num_variables+1:end] .> zero(T))
                    return :Infeasible, x[1:num_variables]
                end
                # Otherwise, start phase two with nice feasible basis.
                for i in 1:num_constraints
                    cB[i] = basic[i] > num_variables ? zero(T) : c[basic[i]]
                end
                continue
            else  # Phase two. We can't improve further, so we are optimal.
                return :Optimal, x[1:num_variables]
            end
        end
        # Calculate how the solution will change when our new variable enters
        # the basis and increases from zero.
        BinvAs = Binv * A[:, entering]
        # Perform a "ratio test" on each variable to determine which will
        # reach zero first.
        leaving = 0
        min_ratio = zero(T)
        for j in 1:num_constraints
            if BinvAs[j] > zero(T)
                ratio = x[basic[j]] / BinvAs[j]
                if ratio < min_ratio || leaving == 0
                    min_ratio = ratio
                    leaving = j
                end
            end
        end
        # If no variable will leave basis, then we have an unbounded problem.
        if leaving == 0
            return :Unbounded, x[1:num_variables]
        end
        # Update solution.
        for j in 1:num_constraints
            x[basic[j]] -= min_ratio * BinvAs[j]
        end
        x[entering] = min_ratio
        # Update basis inverse.
        # Our tableau is [ Binv b | Binv | BinvAs ] and we doing a pivot on the
        # `leaving` row of BinvAs.
        pivot_value = BinvAs[leaving]
        for basis_row in 1:num_constraints
            basis_row == leaving && continue  # All rows except `leaving` row.
            factor = BinvAs[basis_row] / pivot_value
            for basis_col in 1:num_constraints
                Binv[basis_row, basis_col] -= factor * Binv[leaving, basis_col]
            end
        end
        # Finally, the `leaving` row.
        for basis_col in 1:num_constraints
            Binv[leaving, basis_col] /= pivot_value
        end
        # Update variable status flags.
        is_basic[basic[leaving]] = false
        is_basic[entering] = true
        cB[leaving] = phase_one ? zero(T) : c[entering]
        basic[leaving] = entering
    end
end

# end  # module RationalSimplex.
```

Out[17]:
```
simplex
```

Infeasible LP

In [19]:
```
c = [1//1, 0//1]
A = [ 1//1 0//1; -1//1 0//1]
b = [3//1, -2//1];
@time status, x = simplex1(c, :Min, A, b, ['>','>'])
```

```
  0.961453 seconds (2.44 M allocations: 117.780 MiB, 3.70% gc time)
```

Out[19]:
```
(:Infeasible, Rational{Int64}[2//1, 0//1])
```

In [20]:
```
A = [[1,-1]  [0,0]]; b = [3,-2]; c = [1,0]; cstr_types = [1,1]; LP_type = 1;
@time optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
# println("# iterations = ", iters)
```

```
The given LP problem is infeasible!  3.010086 seconds (3.72 M allocations: 148.246 MiB, 0.93% gc time)
```

Bounded and Feasible LP

In [21]:
```
c = [-90//1, -1//1, 0//1, 0//1, 0//1]
A = [  2//1   1//1  1//1  0//1  0//1;
      20//1   1//1  0//1  1//1  0//1;
       2//1   0//1  0//1  0//1  1//1]
b = [ 40//1, 100//1, 3//1]
@time status, x = simplex(c, A, b)
```

```
  0.000087 seconds (64 allocations: 5.938 KiB)
```

Out[21]:
```
(:Optimal, Rational{Int64}[3//2, 37//1, 0//1, 33//1, 0//1])
```

In [22]:
```
A = [[2,20,2]  [1,1,0] [1,0,0] [0,1,0] [0,0,1]]; b = [40,100,3]; c = [-90,-1,0,0,0]; cstr_types = [0,0,0]; LP_type = 
1;
@time optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
println("x₁ = ", optimal_sol[1], ", x₂ = ", optimal_sol[2], ", x₃ = ", optimal_sol[3], "x₄ = ", optimal_sol[4], "x₅ =
 ", optimal_sol[5])
println("optimal cost = ", optimal_cost)
println("# iterations = ", iters)
```

```
  0.032843 seconds (47.82 k allocations: 2.497 MiB)
x₁ = 1.5, x₂ = 37.0, x₃ = 0.0x₄ = 33.0x₅ = 0.0
optimal cost = -172.0
# iterations = 2
```

Unbounded LP

In [23]:
```
c = [-10//1, -12//1, -12//1]
A = [ 1//1 2//1 2//1;
      2//1 1//1 2//1;
      2//1 2//1 1//1]
b = [ 20//1, 20//1, 20//1]
@time status, x = simplex1(c, :Min, A, b, ['>','>','>'])
```

```
  0.000071 seconds (66 allocations: 7.938 KiB)
```

Out[23]:
```
(:Unbounded, Rational{Int64}[0//1, 0//1, 20//1])
```

In [24]:
```
A = [[1,2,2]  [2,1,2]  [2,2,1]]; b = [20,20,20]; c = [-10,-12,-12]; cstr_types = [1,1,1]; LP_type = 1;
@time optimal_sol, optimal_cost, iters = MySimplex(A, b, c, cstr_types, LP_type);
# println("# iterations = ", iters)
```

```
The given LP problem is unbounded!  0.000298 seconds (383 allocations: 31.234 KiB)
```

In [ ]: