Implementation of LUP decomposition and LUPsolve in Julia

Dax Lynch

February 4, 2024

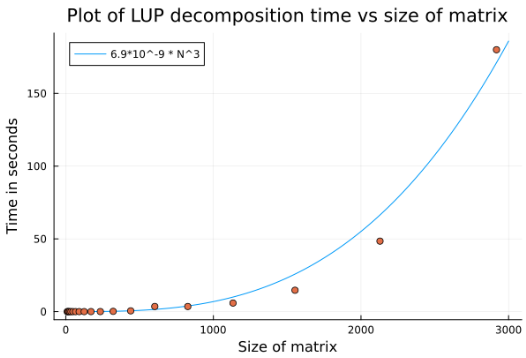
Introduction

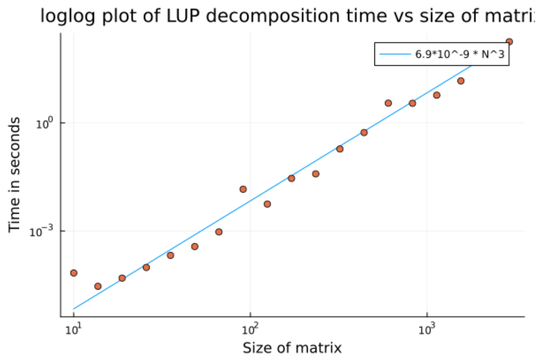
In the code below I implement two functions. One to perform an LUP decomposition on an arbitrary square matrix, and one to solve the system $Ax = b$ using the previous function. This typically involves iterating through the columns, finding the max entry on or below the columns, permuting it so that the max entry is on the diagonal. Then finding the ${L_k}^{-1}$ and L_k such that ${L_k}^{-1} P_k U_{k-1} = L_k U_k$, where $U_0 = A$, U_k has zeros below the diagonal in the first k columns, and L_k has zeros above the diagonals in the first k columns. The tricky part in LUP decomposition is the manipulation of ${L_k}^{-1} P_k {L_{k-1}}^{-1}$ into ${L_k}^{-1} \overline{{L_{k-1}}^{-1}} P_k$.

To compute L, I need to calculate each $\overline{L_k}$ by permuting the off diagonals of every L_k by every permutation matrix after it. To permute the off diagonals, I just permute the matrix with the ℓ_k, and only add in the identity matrix later. Further more instead of keeping each L_k, permuting them every step, and multiplying them at the end, I take advantage of the fact that $L_k L_{k+i}$ is just the identity matrix with ℓ_k and ℓ_{k+i} in their respective spots. So instead of performing a product of N matrices, $L_1 \dots L_N$, I can just keep adding ℓ_k vector to the matrix at each step, swapping an $\mathcal{O}(N^3)$ matrix multiply for an $\mathcal{O}(N)$ column operation. This structure lets me distribute the permutation matrix over the ℓ_k's, so instead of doing $k - 1$ permutations at each step, I only have to do one. Furthermore, instead of doing permutations, I just do row swaps.

Results

My code scaled with $\mathcal{O}(N^3)$ very well, include is a plot with standard axis's, and a plot with log-log axis's. I did a linear regression to find the constant of proportionality, 6.8×10^{-9}. I graphed 19 values, logarithmically placed from 10 to 2900.





Code

```
using Plots

"""
    createI(N)

Returns the identity matrix of size N
"""
function createI(N)
        res = zeros(N, N)
        for i in 1:N
                res[i,i] = 1
        end
        return res
end

"""
    rowSwap!(A, r1, r2)

Performs an inplace rowswap on array A of rows r1 and r2
"""
function rowSwap!(A, r1, r2)
        temp = A[r1, :]
        A[r1, :] = A[r2, :]
        A[r2, :] = temp
end

"""
    computeLUP(A)

Performs an LUP decomposition on a matrix LUP, returning L, U, P
"""
function computeLUP(A)
        N = size(A)[1]
```

```
U = zeros(N,N)
for i in 1:N*N #copies A into U
        U[i] = A[i]
end
Ls = zeros(N,N)
P = createI(N)
for i in 1:N-1
        #This code finds the maximum value in column i, on or
           below the diagonal
        max, maxI = findmax(U[i:N,i]) #findmax in the columns
        maxI = maxI + i - 1

        #Perform the permutation P_k to U, P, Ls, where Ls is
           the matrix of \ell_k's
        rowSwap!(U,i,maxI)
        rowSwap!(P,i,maxI)
        rowSwap!(Ls, i,maxI) #Permute the previous \ell_k's

        Ls[i+1:N, i] = -U[i+1:N, i]/max #Add the new \ell_k to
           Ls

        #Zero the column below. This removes any chance of
           floating
        #point error in the column below U[i,i]. Because
           floating point arithmetic does
        #not always ensure 0 = U[i+k,i] - (U[i+k]/U[i,i]) * U[i,
           i]
        U[i+1:N, i] .= 0

        #Lk = createI(N)
        #Lk[i+1:N, i] = Ls[i+1:N, i]
        #U = Lk * U

        #The code below performs U = Lkinv * U, as does the
           commented out code above, but quicker as it is only
           for the submatrix starting at U[i+1,i+1]. For N =
           1000, the below code takes 1/3 the amount of time of
            the matrix multiplication, and 1/3 less ram.
        # 24.021997 seconds (2.71 M allocations: 15.175 GiB,
           5.48% gc time, 7.98% compilation time)
        #vs
        # 8.269111 seconds (2.82 M allocations: 10.365 GiB,
           16.32% gc time, 14.50% compilation time)

        for j in i+1:N  #I perform this operation using columns
           as julia is column major,
          U[i+1:N, j] += Ls[i+1:N, i] * U[i, j]
        end
```

```
                #For each column j, below the i'th row, add to it the i'
                    th column of Ls, times the j'th element in the i'th
                    row.

        end

        return (-Ls + createI(N)), U, P
end

"""
    LUPsolve(A, b)

Solves the matrix vector equation, Ax = b, by first performing an LUP
    decomposition, then performing backwards then forwards substitution.
     returns the vector x
"""
function LUPsolve(A, B)
N = size(A)[1]

   L, U, P = computeLUP(A)
   b = P*B
   b[1] /= L[1,1]
   for i in 2:N #forward solve
           b[i] -= sum(L[i, 1:i-1] .* b[1:i-1]) #b[i] is equal to itself
                minus the dot product of the entries above it by the
               entries in the to the left L[i,i]
           b[i] /= L[i,i]
   end
   b[N] /= U[N,N]
   for i in reverse(1:N-1) #backwards solve
           b[i] -= sum(U[i,i+1:N] .* b[i+1:N]) #b[i] is equal to itself
               minus the dotproduct of the entries below it by the
               entries to the right of U[i,i]
           b[i] /= U[i,i]
   end
return P, b
end

"""
    myTrans(A)

returns the transpose of an array A
"""

function myTrans(A)
N = size(A)[1]
temp = zeros(N,N)
```

```julia
    for i in 1:N
            temp[i,i] = A[i,i]
            for j in i+1:N
                    temp[i,j] = A[j,i]
                    temp[j,i] = A[i,j]
                 end
    end
return temp
end

N = 100
A = rand(N,N)
L, U, P = computeLUP(A)
@assert isapprox(L*U, P*A)
@time L, U, P = computeLUP(A)


B = rand(N, N)
A = myTrans(B) * B + createI(N)
b = rand(N)
P, x = LUPsolve(A, b)

@assert isapprox(A*x, b)
@time P, x = LUPsolve(A, b)

#Below is how I computed times and plots
#for N in range(log10(10),log10(4000),20)
#       println(round(Int,10^N))
#       b = round(Int, 10^N)
#       A = rand(b,b)
#       @time L, U, P = computeLUP(A)
#end

#x =  10 .^ range(log10(10),log10(4000),20) #these are the
    logarithmically spaced sizes
#y = [0.000068 0.000029 0.000049 0.000097 0.000210 0.000369 0.000941
    0.014399 0.005584 0.028830 0.038493 0.189202 0.539899 3.540114
    3.508246 5.894943 14.696663 48.448576 179.996518 669.530503] These
    are the recorded times

#x3 = 6.88636709e-09 * range(10,3000,1000) .^ 3

#plot(range(10,3000,1000), x3, label="6.9*10^-9 * N^3")
#scatter!(x,y,label="", legend=true,xlabel="Size of matrix",ylabel="Time
     in seconds",title="Plot of LUP decomposition time vs size of matrix
    ")
#savefig("TimevsN.png")
```