CIS 510 HW 1 Submission

Scott Lambert
10/12/2022

Prompt

Not all square matrices admit an LU-factorization, but they do admit an LUP-factorization $PA = LU$, where P is a permutation matrix. Develop the Julia script *my_solvers.jl* to include the function *computeLUP*() that does an LUP-decomposition of a square matrix A using partial pivoting. Next add to the script a function $LUPsolve()$ that solves $Ax = b$ by first computing an LUP-factorization (i.e it calls *computeLUP*()) and then does forward and backward substitution. Include in your script tests that confirm accuracy of your functions using the matrix $A = B^T B + I$, where B =rand(N, N) and I is the $N \times N$ identity matrix. You can set vector $b = rand(N, 1)$. Time the two (albeit unoptimized) functions using the @*time* macro using $N \times N$ matrices with $N = 10, N = 100$ and $N = 1,000$.

Ideas for figure: plot time as a function of N.
Ideas for conclusions: Computing LUP and then doing back substitution is equivalent to the work/complexity of Gaussian Elimination (i.e. $O(N^3)$). Is this generally what you see?

Solution

See the end of this file for the full `my_solvers.jl` source. Note that I used the function names `solvePLU` and `computePLU` instead of `LUPsolve` and `computeLUP`, respectively. The top-level function calls that generated the data in this writeup are at the very end of the file.

Accuracy of PLU factorization and solver

Shown below are plots of the error of the PLU factorization algorithm `computePLU` and the linear solver `solvePLU`. For each blue data point on the plots, an NxN matrix A of the form $A = B^T B + I$ was generated as per the problem statement. This matrix was then used as an argument to `computePLU` and `solvePLU`, and errors were computed as described in the plot captions. In the case of the `solvePLU` plot, the vector b was a randomly generated Nx1 column vector. Random matrices were generated using the `rand` function in the Julia Base. Linear fits created using `linear_fit` from the CurveFit package are also shown.

Since new random NxN matrices were passed to the solver for each value of N, there is some stochasticity in the results. Results are only shown for a single set of trials, so rerunning the code may produce slightly different plots, but the fitted line slopes were found to stay within about ± 0.5 of 5 and 4 for the factorization and solver plots, respectively. Note how tiny the errors are: for a 1000x1000 matrix ($\log_{10} N = 3$), the matrix norm of the error is 10^-20. Nonetheless, it is interesting and somewhat alarming to note that the error scales as nearly $\mathcal{O}(N^5)$ (corresponding to a fit slope of 5) for the factorization algorithm and $\mathcal{O}(N^4)$ for the solver.

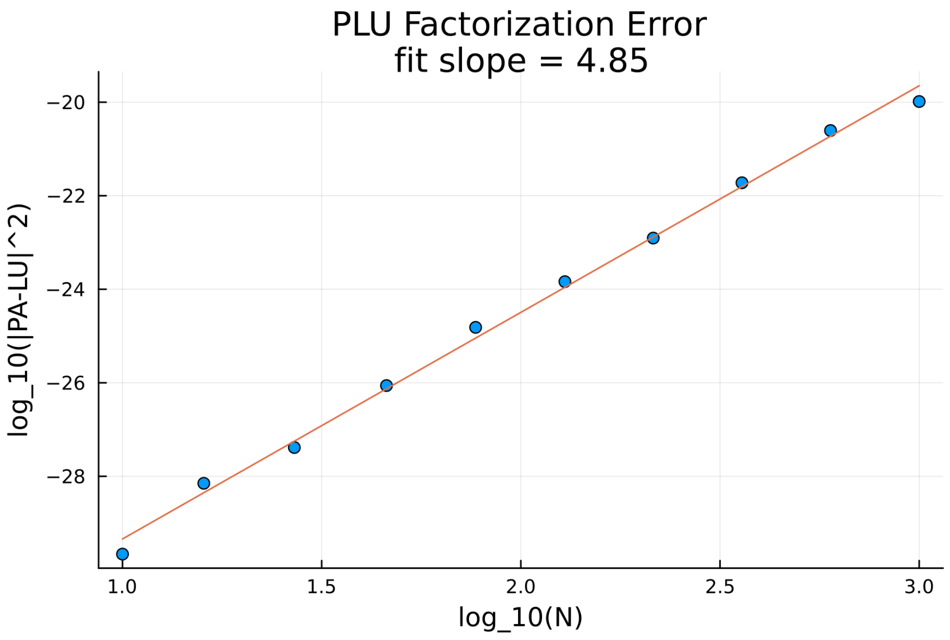

Figure 1: Error of the PLU factorization algorithm, given as the base-10 log of the squared matrix norm $|PA - LU|^2$. Here vertical bars denote the Frobenius norm (square root of sum of squares of all elements). Blue dots are computed errors and the line is a linear fit.

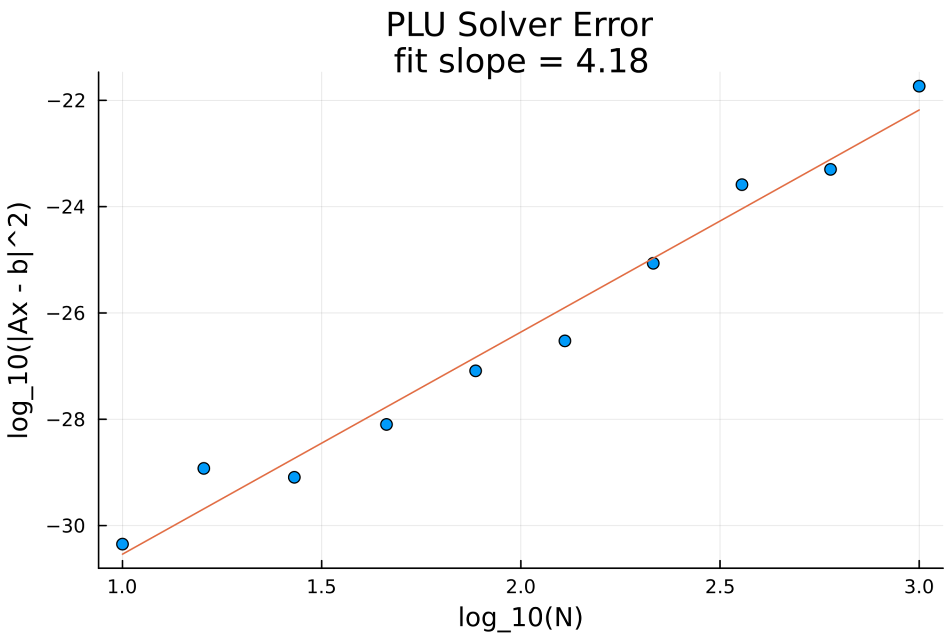

Figure 2: Error of the PLU solver algorithm, given as the base-10 log of the vector norm $|Ax - b|^2$. Here vertical bars denote the Euclidean norm. Blue dots are computed errors; the orange line is a linear fit.

To finish this demonstration of the PLU factorization algorithm's accuracy, computed PA and LU for $N = 4$ are shown below. The matrices are identical to the accuracy shown, and the error (given by the squared Frobenius norm $|PA - LU|^2$) was too small to be printed in this case (see the plots above for more error data).

```
PA =
4 4  Matrix{Float64}:
 2.6258    1.18299   0.818426  0.99455
 1.18299   2.0205    0.685413  0.885197
 0.818426  0.685413  1.50502   0.566379
 0.99455   0.885197  0.566379  1.79291
LU =
4 4  Matrix{Float64}:
 2.6258    1.18299   0.818426  0.99455
 1.18299   2.0205    0.685413  0.885197
 0.818426  0.685413  1.50502   0.566379
 0.99455   0.885197  0.566379  1.79291
 Error = 0.0000e+00
```

Performance of the PLU factorization

The `@elapsed` macro was used to test the PLU factorization algorithm's performance. Test NxN matrices were generated as given by the problem statement and in the accuracy tests above; these were passed to the function `solvePLU(A,b)` along with a random Nx1 vector b. Note that the slope of the linear fit is approximately 3, indicating that the execution time scales as approximately $\mathcal{O}(N^3)$. Execution times ranged from less than one millisecond for a 10x10 matrix ($\log_{10} N = 1$) to 100 seconds for a 1000x1000 matrix.

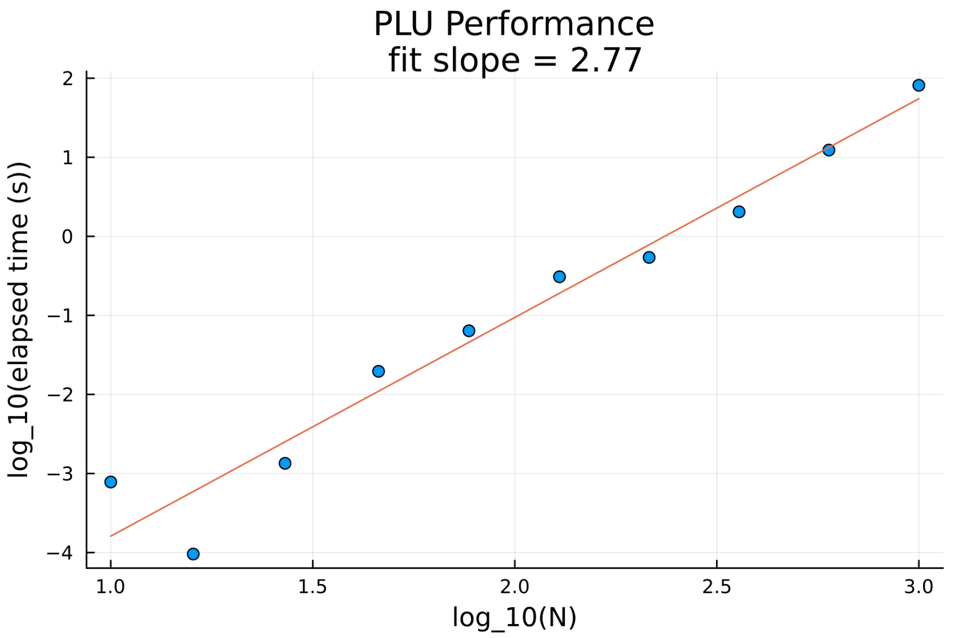

Figure 3: Performance of the PLU solver algorithm, given as the base-10 log of the time in seconds required to run `solvePLU(A,b)` for a matrix of size N. Blue dots are computed errors and the orange line is a linear fit. Note that the slope is nearly 3, which is consistent with the complexity scaling as $\mathcal{O}(N^3)$.

Conclusions

The errors in the outputs of `solvePLU` and `computePLU` are tiny relative to the size of the matrix elements; this indicates that these functions accurately compute PLU factorizations and use them to solve linear systems. Execution time for `solvePLU` scales as approximately $\mathcal{O}(N^3)$, consistent with theoretical predictions from an analysis of Gaussian elimination. The errors of these algorithms (as measured above) also seem to scale as integer powers of N, with exponent 5 for the PLU factorization and 4 for the solver.

Full my_solvers.jl file

```
using Plots
using CurveFit
using Printf

function backsub(U,b)
"""
    backsub(U,b)
Solve Ux = b via back-substitution and return the solution x.

Notes:
U must be an upper-triangular matrix of size N with nonzero elements
along the diagonal; b must be a column vector of length N
"""

    N = size(b)[1]
    x = zeros(N,1)
    for i = 0:N-1
        x[N-i] = (1/U[N-i,N-i])*(b[N-i]-sum(U[N-i,N-i+1:N].*x[N-i+1:N]))
    end
    return x
end

function forwardsub(L,b)
"""
    forwardsub(L,b)
Solve Lx = b via forward-substitution.
L must be a lower-triangular matrix of size NxN with nonzero elements
along the diagonal; b must be a column vector of length N
"""
    N = size(b)[1]
    x = zeros(N,1)
    for i = 1:N
        x[i] = (1/L[i,i])*(b[i]-sum(L[i,1:i-1].*x[1:i-1]))
    end
    return x
end

function pivot_matrix(A,k)
"""
    pivot_matrix(A,k)
Compute the permutation matrix P such that left-multiplying A by
P puts the largest-magnitude element in the kth column at or below the
diagonal on the diagonal

returns: permutation matrix P
"""
    N = size(A)[1]
    # Get elements of kth column on and below diagonal
    col = A[:,k]
    # Zero the first k-1 elements of col so that they won't be picked
    # for the pivot
    col[1:k-1] .= 0
    # Find index j of maximum element
    j = findmax(abs.(col))[2][1]
    # Create permutation matrix swapping rows j and k
    P = identity(N)
    P[k,:],P[j,:] =  P[j,:],P[k,:]
    return P
end

function identity(N)
"""
    identity(N)
return the NxN identity matrix.
"""
    I = zeros(N,N)
    for i = 1:N
        I[i,i] = 1
    end
    return I
end

function Lstep(U, k)
"""
    Lstep(U, k)
Compute the matrix Ls required to eliminate the below-diagonal elements of
column k from the matrix U via the multiplication Ls*U
Assumes U[k,k] != 0
"""
    N = size(U)[1]
    # Vector to be inserted into I to form Lk:
    col = zeros(N,1)
    col[k] = 1
    col[k+1:N] = -U[k+1:N,k]./U[k,k]
    Ls = identity(N)
    Ls[:,k] = col
    return Ls
end

function LstepInv(Ls,k)
"""
    LstepInv(Lstep, k)
Compute the inverse of the row-reduction matrix Lstep given
that the column containing the elimination coefficients is k.
Lstep must be of the form returned by Lstep(), ie an identity
matrix but containing a single column with possibly-nonzero
coefficients below the diagonal.
"""
    N = size(Ls)[1]
    LsInv = identity(N)
    LsInv[k+1:N,k] = -Ls[k+1:N,k]
    return LsInv
end

function computeLU(A)
"""
    computeLU(A)
compute the LU factorization of a matrix A.
A must be a square matrix that admits an LU factorization (ie,
must be a matrix such zero-valued pivot elements don't appear
during Gaussian elimination).
"""
    N = size(A)[1]
    L = identity(N)
    U = A
    for k = 1:N
        Ls = Lstep(U,k)
        U = Ls*U
        L = L*LstepInv(Ls,k)
    end
    return L,U
end

function computePLU(A)
"""
    computePLU(A)
Determine P, L, and U such that PA = LU where P is a permutation matrix,
L is lower-triangular, and U is upper-triangular. A must be a square
matrix.
"""
    N = size(A)[1]
    U = A
    L = identity(N)
    P = identity(N)
    for k = 1:N-1
        Ps = pivot_matrix(U,k)
        U = Ps*U
        Ls = Lstep(U,k)
        U = Ls*U
        P = Ps*P
        L = Ps*L*Ps*LstepInv(Ls,k)
    end
    return P, L, U
end

function testPLU(N)
"""
    testPLU(N)
Test the accuracy of the PLU factorization algorithm.
Generates a random NxN matrix B, computes the PLU factorization
of transpose(B)*B + I using computePLU, and finds the
square of the matrix norm of the difference of PA and LU
"""
    B = rand(N,N)
    A = transpose(B)*B+identity(N)
    P,L,U = computePLU(A)
    return sum((P*A - L*U).^2)
end

function solvePLU(A,b)
"""
    solvePLU(A,b)
Solves Ax = b via PLU factorization. A is NxN, b is Nx1
"""
    P,L,U = computePLU(A)
    y = forwardsub(L,P*b)
    x = backsub(U,y)
    return x
end

function testsolvePLU(N)
    """
        testPLU(N)
    Test the accuracy of solvePLU.
    Solves Ax = b with solvePLU and outputs the squared error |Ax - b|^2.
    """
        B  = rand(N,N)
        A = transpose(B)*B + identity(N)
        b = rand(N,1)
        x = solvePLU(A,b)
        return sum((A*x - b).^2)
    end

function showPLUExample(N)
"""
    showPLUExample()
Generate a matrix A and show P, L, and U as calculated
by computePLU(A). Also show the products PA and LU and
show the squared error |PA - LU|^2
"""
    B = rand(N,N)
    A = transpose(B)*B+identity(N)
    P,L,U = computePLU(A)
    println("PA = ")
    display(P*A)
    println("LU = ")
    display(L*U)
    @printf("Error = %.4e",sum((P*A-L*U).^2))
end

function plotPLUTestResults(N,data,title,ylabel)
"""
    plotPLUTestResults()
Plot results of PLU algorithm testing (convenience function).
N = array of matrix sizes (scalars, not tuples; matrices are square)
data = vector of errors or execution times corresponding to entries of N
"""
    xplot = log.(N)./log(10)
    yplot = log.(data)./log(10)
    p = scatter(xplot,yplot,
        xlabel = "log_10(N)", ylabel = ylabel,legend=false)
    a0,a1 = linear_fit(xplot,yplot)
    plot!(p,xplot,a0 .+ a1 .* xplot)
    title!(@sprintf("%s\nfit slope = %.2f",title,a1))
    display(p)
end

function plotsolvePLUAccuracy(ntrials,maxpow10)
"""
    plotsolvePLUAccuracy()
Solve Ax = b, where A is an NxN matrix, using solvePLU for different
values of N and plot the error |Ax-b|^2 as a function of N
"""
    # N-values are logarithmically spaced between 10 and 10^maxpow10
    N = Int.(floor.(10 .^range(1,maxpow10,ntrials)))
    errors  = zeros(ntrials)
    for (i,n) in enumerate(N)
        @printf("\n i = %d, n = %d",i,n)
        errors[i] = testsolvePLU(n)
    end
    plotPLUTestResults(N, errors, "PLU Solver Error","log_10(|Ax - b|^2)")
    #=
    p = scatter(log.(N),log.(errors),title = "PLU solver error",
        xlabel = "log(N)", ylabel = "log(|Ax - b|^2)")
    a0,a1 = linear_fit(log.(N),log.(errors))
    plot!(p,log.(N),a0 .+ a1 .* log.(N))
    title!(@sprintf("PLU Accuracy\nfit slope = %.2f",a1))
    display(p)
    =#
end

function plotPLUAccuracy(ntrials,maxpow10)
    """
        plotPLUAccuracy()
    Compute the squared matrix norm of
    the difference PA - LU for the LU factorization of A at various
    N and plot the data
    """
        # N-values are logarithmically spaced between 10 and 10^maxpow10
        N = Int.(floor.(10 .^range(1,maxpow10,ntrials)))
        errors  = zeros(ntrials)
        for (i,n) in enumerate(N)
            @printf("\n i = %d, n = %d",i,n)
            errors[i] = testPLU(n)
        end
        plotPLUTestResults(N, errors, "PLU Factorization Error","log_10(|PA-LU|^2)")
        #=
        p = scatter(log.(N),log.(errors),title = "PLU factorization error",
            xlabel = "log(N)", ylabel = "log(|PA-LU|^2)")
        a0,a1 = linear_fit(log.(N),log.(errors))
        plot!(p,log.(N),a0 .+ a1 .* log.(N))
        title!(@sprintf("PLU Accuracy\nfit slope = %.2f",a1))
        display(p)
        =#
    end

function plotPLUPerformance(ntrials,maxpow10)
"""
    plotPLUPerformance()
Plot the time required to solve Ax = b, where A is an NxN matrix,
at different values of N.
"""
    # N-values are logarithmically spaced between 10 and 10^maxpow10
    N = Int.(floor.(10 .^range(1,maxpow10,ntrials)))
    times  = zeros(ntrials)
    for (i,n) in enumerate(N)
        @printf("\n i = %d, n = %d",i,n)
        B = rand(n,n)
        A = transpose(B)*B + identity(n)
        b = rand(n,1)
        times[i] = @elapsed solvePLU(A,b)
    end
    # Plot the performance data
    plotPLUTestResults(N, times, "PLU Performance","log_10(elapsed time (s))")
    #=
    p = scatter(log.(N),log.(times),
    xlabel="log(N)", ylabel="log(Elapsed time (s))")
    # Fit a line to the data
    a0,a1 = linear_fit(log.(N),log.(times))
    plot!(p,log.(N),a0 .+ a1 .* log.(N))
    title!(@sprintf("PLU Performance\nfit slope = %.2f",a1))
    display(p)
    =#
end


N = 4
ntrials = 10
maxpow10 = 3

showPLUExample(N)
plotPLUAccuracy(ntrials,maxpow10)
plotsolvePLUAccuracy(ntrials,maxpow10)
plotPLUPerformance(ntrials,maxpow10)
```