Problems 1 and 4 have us check the run times of a few different algorithms as they are run on increasingly large $N \times N$ matrices. It is worth nothing that in Figure 1 computeLUP and LUPsolve are my naive implementations of an LU factorization and back/forward substitution to solve $Ax = b$. Now, each algortihm is tested on random $N \times N$ matrices but juliaLUsolve and juliaCGsolve were both run on symmetric, positive definite matrices. I expected CG to perform the best (especially as N increased) but it seemed that once you have an LU-decomposition you can get the solution pretty quick. This is because you are basically ignoring the most computationally expensive part. We know back/forward substitution is $O(N^2)$ and by omitting the time it takes to find $LU = A$ it feels like we are seeing a $N \times N$ system solved in $O(N^2)steps$. Now I am not sure how much I actually trust that my LUPsolve runs faster than julia's optimized functions but I would be curious to see how they optimize back substitution. Additionally, though not recorded in Figure 1, julias native LU function caught me of guard with how fast it runs. My LU-decomposition code took about 62 seconds on a 1000×1000 matrix but julia could compute a factorization in about 0.05 seconds. This seems like a huge speed up and I would like to know what is going on there. Figure 2 which shows the time it takes to decompose a 100×100 tridiagonal matrix into its LU-decomposition along with how long it takes to solve the resulting system. The decomposition time for the dense case seems to agree with the measured time for my own LU-decomposition when $N = 100$ but at $N = 1000$ my algorithm slows way down while julia's seems to slow only slightly. The final bit of information from Figure 2 shows significant performance gains while factoring a sparse matrix. The sparse matrix also displays faster solve times but I thought thebiggest difference was in the decomposition time and the fact that the sparse matrix required only 40 bytes to store while the dense matrix required 80,000 bytes.

N	computeLUP	LUPsolve	juliaLUsolve	juliaCGsolve
10	$8.142e^{-3}$	$5.000e^{-6}$	$1.700e^{-5}$	$2.400e^{-5}$
100	$2.845e^{-2}$	$6.400e^{-5}$	$1.180e^{-4}$	$3.552e^{-3}$
1000	$6.273e^{1}$	$4.100e^{-3}$	$1.309e^{-2}$	$6.335e^{-1}$

Figure 1: Run time data for my solvers compared to julia's for $N \times N$ matrices.

	LU-Decomposition	Solve
dense	$2.723e^{-2}$	$5.620e^{-4}$
sparse	$7.750e^{-4}$	$6.200e^{-5}$

Figure 2: Comparison of decomposition and solve times for dense and sparse matrices.

MySolvers.jl

```
using LinearAlgebra
using Printf

"""
    find_pivot(a)

Locate the element of maximal absolute value in the one-dimensional arra

# Examples
```jldoctest
julia> A = [ 1 2 -3]
julia> j = find_pivot(A)
```
"""
function find_pivot(a, k)
    N = length(a)
    p = 0

    a = broadcast(abs, a)

    for i = k:N
        if abs(p) < abs(a[i])
            p = a[i]
        end
    end

    q = findall(x -> x==p, a)
    q = q[1]

    return q
end

"""
    computeLUP(a)

Compute and return LUP factorization of square matrix a.

Examples
```jldoctest
julia> A = rand(3,3)
```
```

```
julia> (L, U, P) = LUPsolve(A)
```
```
"""
function computeLUP(A)
    # TODO: develop to include partial pivoting
    N = size(A)[1]

    Id      = Matrix{Float64}(I, N, N)
    ell     = copy(Id)
    ell_inv = copy(Id)
    Atilde  = copy(A)
    L       = copy(Id)
    P       = copy(Id)
    P_ij    = copy(Id)


    j = find_pivot(Atilde[:,1], 1)

    if j != 1
        (P_ij[j,: ], P_ij[1, :]) = ( P_ij[1, :], P_ij[j,: ] )
        (Atilde[j, 1:N ],Atilde[1, 1:N]) = ( Atilde[1, 1:N], Atilde[j,
    end

    for k = 1:N-1  # marching across columns


        ell .= Id
        ell_inv .= Id


        for i = k+1:N
            ell[i,k] = -Atilde[i,k] / Atilde[k,k]
            ell_inv[i,k] = Atilde[i,k] / Atilde[k,k]
        end

        Atilde .= ell * Atilde
        L .= L * ell_inv

        j = find_pivot(Atilde[:,k+1], k+1)

        if j != k+1
            (P_ij[j,: ], P_ij[k+1, :]) = ( P_ij[k+1, :], P_ij[j,: ] )
            (L[k+1,1:k ], L[j, 1:k]) = ( L[j, 1:k], L[k+1,1:k ] )
```

```
            (Atilde[j, : ],Atilde[k+1, :]) = ( Atilde[k+1, :], Atilde[j
        end



    end
    U = Atilde
    P = P_ij

    return (L, U, P)
end

"""
    LUPsolve(A, b)

Solve the matrix equation Ax=b using LU factorization.

# Examples
```jldoctest
julia> A = rand(3,3)
julia> b = rand(3,1)
julia> x = LUPsolve(A, b)
```
"""
function LUPsolve(A, b)

    N = size(L)[1]

    L, U, P = computeLUP(A)

    b = P*b

    y = forward_sub(L, b)
    x = backward_sub(U, y)

    return x

end

"""
    backward_sub(A, b)
```

```
Solve the matrix equation Ax=b where A is upper triangluar.

# Examples
```jldoctest
julia> A = [1 2 3;
            0 4 5;
            0 0 6]
julia> x = backward_sub(A, b)
```
"""
function backward_sub(A, b)
    N = size(A)[2]
    for i = 1:N # march along columns
        k = N+1-i # backwards

        b[k] = b[k] / A[k,k] # normalize the pivot
        for j = 1:k-1

            b[j] = b[j] - b[k] * A[j,k]
        end
    end
    return b
end

"""
    forward_sub(A, b)

Solve the matrix equation Ax=b where A is lower triangluar.

Examples
```jldoctest
julia> A = [1 0 0;
            2 3 0;
            4 5 6]
julia> x = forward_sub(A, b)
```
"""
function forward_sub(A, b)
    N = size(A)[2]

    for i = 1:N # march along columns
```

```
        b[i] = b[i] / A[i,i]
        for j = i+1:N

            b[j] = b[j] - b[i]*A[j,i]
        end
    end
    return b
end
```

SparseDenseTests.jl

```
using LinearAlgebra
using SparseArrays

function comp_piv_matrices(p,q,r)

    N = length(p)

    Id = sparse(1:N,1:N,ones(N),N,N)
    P = copy(Id)
    Q = copy(Id)

    for i = 1:N
        P[i,:] = Id[p[i],:]
        Q[:,i] = Id[:,q[i]]
    end

    R = sparse(1:N,1:N,r,N,N)

    return (P,Q,R)

end

N = 100
A = zeros(N, N)
b = ones(N, 1)

for i = 2:N-1
    A[i,i] = -2
    A[i, i-1] = 1
    A[i, i+1] = 1
end
A[1, 1] = -2
A[1, 2] = 1
A[N, N] = -2
A[N, N-1] = 1

# LU decomposition using native julia function
@time F = lu(A)

@time x = F.U \ (F.L \ b)
```

```
@assert A*x     b

# Sparse A

Id = 1:N
Jd = copy(Id)
Vd = -2 * ones(N)

Adiag = sparse(Id, Jd, Vd, N, N)

Il = 2:N
Jl = 1:N-1
Vl = ones(N-1)

Iu = 1:N-1
Ju = 2:N
Vu = ones(N-1)

Asparse = Adiag + sparse(Il, Jl, Vl, N, N) + sparse(Iu, Ju, Vu, N, N)

# LU decomposition using native julia function
@time F = lu(Asparse)

(P,Q,R) = comp_piv_matrices(F.p,F.q,F.Rs)

#time actual solve
@time x = Q*(F.U\(F.L\(P*R*b)))

#time factorization plus solve together:
xMod = Asparse\b

@assert xMod     x

@show sizeof(Asparse)
@show sizeof(A)
```

NativeSolvers.jl

```
using LinearAlgebra
using IterativeSolvers

N = 1000
B = rand(N,N)
A = B*B'

b = ones(N,1)

# LU solve
F = lu(A)

@time x = F.U \ (F.L \ b)
@assert A*x     b

# CG solve
@time y = cg(A, b; tol=1e-9, maxiter=3*N)
@assert A*y     b
```