Report on the project F6_Sparse

Why Sparse

A matrix is a n x m structure, where n is the number of rows and m the number of columns, storing for each cell a number.
A characteristic of a matrix is its sparsity, namely the total number of zero valued elements divided by the total number of elements.
A dense matrix is a matrix with a low sparsity, and therefore a high number of non-zero entries. On the other hand, a sparse matrix is a matrix with a low number of non-zero entries; it's a matrix composed for the most part of zeros.
Typically, a matrix is stored in memory by saving each value in its corresponding position of the matrix. However, since in a sparse matrix most of the entries are zero, we can take advantage of this knowledge and store only the non-zero elements, assuming the non-stored entries are all zeros. This is what is called a sparse representation.
In Julia it is yet implemented a type, SparseMatrixCSC, that translates the sparse representation.

SparseMatrixCSC

In [ ]:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

SparseOperator

In [1]:

```
include("Ban.jl")
using .BAN
```

In [2]:

```
using SparseArrays
A = [one(Ban) zero(Ban);zero(Ban) zero(Ban)]
sparse(A)
```

Out[2]:

```
2×2 SparseMatrixCSC{Ban, Int64} with 1 stored entry:
 α^0(1.0 + 0.0η^1 + 0.0η^2)                          ⋅
              ⋅                                     ⋅
```

The sparse( ) method is a method of the Julia SparseArrays library that allows to translated a Matrix structure into a SparseMatrixCSC structure.
This method works by checking, for each cell of the original matrix, if its entries is equal to 0 or not; if it is, the corresponding SparseMatrixCSC will be coherently updated.
In our case since the '==' operator was yet reimplemented in the Ban library provided to us, we didn't have to extend this function: when the program will try to execute the sparse( ) method it will make a comparison between the Ban entry of the matrix and a zero Ban.

In [3]:

```
Base.:(==)(a::Ban, b::Ban) = (a.p == b.p && a.num == b.num)
```

However, even though the operator works properly,we had to rewrite the show()

Old Show

In [ ]:

```
function _show(io::IO, a::Ban)

    print(string("α^",a.p,"(",a[1]))
    for i=2:SIZE
        if a[i] >= 0
            print(string(" + ", a[i], "η^$(i-1)"))
        else
            print(string(" - ", -a[i], "η^$(i-1)"))
        end
    end
    print(")")
end
```

In the previous implementation of the show for the Bans, the redirection of the output was done only towards the standard output.
This presented a problem with the print of the SparseArrays library: as it is implemented at the moment, for each element of the SparseMatrixCSC it will be called the function *sprintf* , that will in turn call *@printf* , which returns the output formatted as a string; but since the *@printf* works in different IO than the one of the Bans, this caused the function to return an empty string. The Bans were not displayed and an error occurs during the show().
The problem has been resolved by correting the original show in the Ban library. This resolved also the problem of the multiple shows that occurs while using the Ban type.

New Show

In [ ]:

```
function _show(io::IO, a::Ban)
    print(io, string("α^",a.p,"(",a[1]))
    for i=2:SIZE
        if a[i] >= 0
            print(io, string(" + ", a[i], "η^$(i-1)"))
        else
            print(io, string(" - ", -a[i], "η^$(i-1)"))
        end
    end
    print(io, ")")
end
```

Why use Cholesky?

Let's suppose to have a linear system:
$Ax = b$, in which A is a nxn matrix, b a vector in $\Re^n$ and x a vector in $\Re^n$.

The typical way to resolve this system is to apply the substitution method , namely use one equation to solve for one variable, and then substitute that expression into another equation to solve for another variable,and so on so forth until it remains a single expression,in only one variable, that can be resolved. And then we use backward-substitution to resolve the previous equations.

This approach is computationally heavy, and could be simplified through the use of the matrix factorization. The Cholesky decomposition is one of the most efficient way to correctly factorize a matrix to solve a linear system.

Cholesky Decomposition

Under the assumption that the matrix A is a Hermitian positive-definite matrix, the matrix can be factorized in the following form:

$$A = LL^T$$

where L is a lower triangular matrix, and L^T denotes the transpose of L.

The linear system can therefore be rewritten as follows:

$$LL^T x = b$$

And, if we define $L^T x = y$, the system can be solved with two triangular substitution:

$$\begin{cases} Ly = b \\ L^T x = y \end{cases}$$

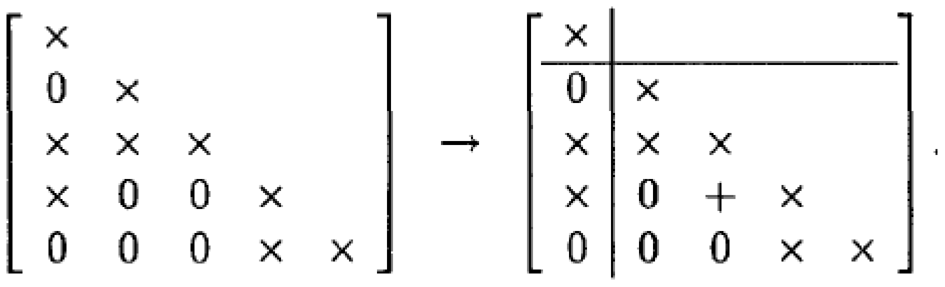
Cholesky Decomposition, pseudo-code

In [ ]:

```
L = spzeros(T, n, m)

for i in 1:n
    L[i, i] = sqrt( norm(M[i, i]) )
    for j in i+1:m
        L[j,i] = M[j, i]/L[i,i]
        for k in i+1:j
            M[j,k] = M[j,k] - (L[j,i]*L[k,i])
        end
    end
end
```

Sparse Cholesky Factorization

Cholesky factorization of a sparse matrix usually produces fill-in(represented as + in the following image); that is,some lower triangular locations in the factor L contain nonzero elements even though the same locations in the original matrix M are zero.



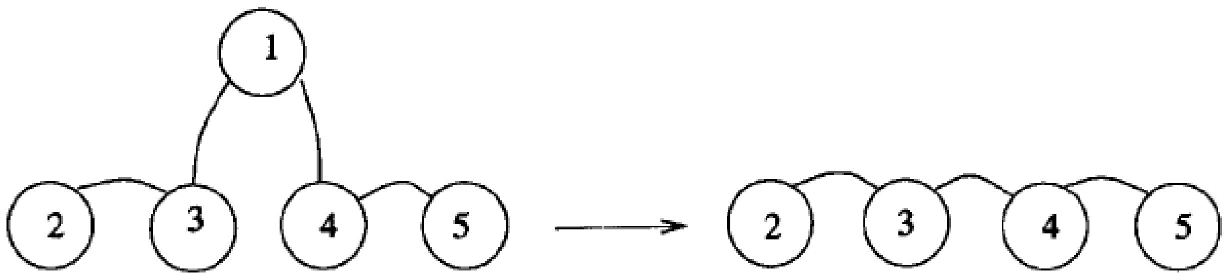
We can apply some control over the amount of fill-in through our choice of the row/column ordering in M, that is, our choice of the permutation matrix P.
We would like to keep fill-in small for two reasons:

- The fill-in requires additional storage</p>
- The extra nonzeros add to the cost of updating the remaining matrix and also the cost of the triangular substitutions with L and L^T that are needed to recover the solution vector</p>

Therefore, the idea is to find an ordering that minimizes the fill-in

Minimum Ordering Degree Algorithm

The term degree refers to the number of nonzero elements in a column of the matrix, excluding the diagonal. At each step, the algorithm examines the remaining matrix and selectes the diagonal element with the minimum degree (let's suppose the index of the diagonal element is **k**). We permute by swapping rows 1 and k and columns 1 and k of the remaining matrix, ending with the diagonal element (k,k) in position (1,1); then, we apply a factorization step,and finally perform a gaussian elimination of the first row and column.
This type of ordering is motivated by the fact that, during the factorization step, if a pivot element (k,k) has degree **d**, the updated matrix will contain **d^2** nonzeros. It makes sense to choose the pivot so that d is as small as possible.
However, instead of alternating the step of permutation and factorization, which results to be inefficient because, for example, it doesn't fully exploit the cache, we opt for first obtain the permutation of the entire matrix, and then factorize.
In order to do so, we construsct a graph and find the node with minimum degree. The index of the node represents a step of the permutation. By modifying the graph, namely by eliminating the chosen node and updating the edges, we are able to keep track of the new non-zero elements in the matrix and choose the next node with minimum degree, without performing any step of factorization.



We repeat the process until we obtain the entire array of permutation.

Minimum Local Fill Algorithm

Another algorithm for the ordering of the matrix is called minimum local fill.
The minimum local fill works on the adjacency graph: at each stage of the elimination it makes a prediction of the number of new edges that would be created if we were to eliminate this node (namely, the fill-in). The elimination node is then chosen as the one that obtains the minimum fill-in.

Cholesky Implementation for SparseMatrixCSC of Ban

Struct BanFactor

The structure BanFactor has been implemented as a wrapper to contain the Cholesky, and (explained later in the report) LDLT representation.
Other than the L,D,Lt,P(Permutation vector,where needed), it contains a flag that represents if the factorization has been successfull or not, and if its has been which type of factorization it holds.

In [3]:

```
struct BanFactor
    L::AbstractMatrix
    D::AbstractMatrix
    Lt::AbstractMatrix
    P::Vector

    #  0    - LDLT
    #  1    - Cholesky
    #  2    - NULL
    flag::UInt8

    # not success costructor
    BanFactor( _flag::UInt8 = UInt8(2) ) = new( Array{Float64}(undef, 0, 2), Array{Floa
t64}(undef, 0, 2), Array{Float64}(undef, 0, 2), [], _flag)

    # LDLT constructor
    BanFactor(_L::AbstractMatrix, _D::AbstractMatrix, _Lt::AbstractMatrix, _P::Abstract
Vector, _f::UInt8 =  UInt8(0) )= new( _L, _D, _Lt, _P, _f)
    BanFactor(_L::AbstractMatrix, _D::AbstractMatrix, _Lt::AbstractMatrix, _f::UInt8 =
UInt8(0)) = new( _L, _D, _Lt, [], _f)

    # Cholesky constructor
    BanFactor( _L::AbstractMatrix, _Lt::AbstractMatrix, _P::AbstractVector, _f::UInt8 =
UInt8(1)) = new( _L, Array{Float64}(undef, 0, 2), _Lt, _P,  _f)
end
```

Why we need this structure?

The structure 'Cholesky' of julia returns a type that doesn't include a field for the permutation, therefore we weren't able to extend this type for our purposes.
An even worst situation occurs with the LDL^T (whose discussion will be later in the document) since in julia it is implemented only for the 'SymTridiagonal' type; also, inside the implementation, it will be made us of specific characteristic of the tridiagonals.
Since it was impractical to re-extend both the two libraries, we decided to adopt this approach.

In [4]:

```
using LinearAlgebra
using SparseArrays
using Laplacians
using Random
using LightGraphs
using AMD

function _minimum_degree( G::Graph )
    adj_M = LightGraphs.LinAlg.CombinatorialAdjacency( LightGraphs.LinAlg.adjacency_mat
rix(G) )
    degrees = LightGraphs.LinAlg.degrees(adj_M)

    value, index = findmin(degrees)

    return index
end

function _minimum_local_degree(G::Graph, size::Int64)
    min_count = Base.Inf64
    index = -1
    for i in 1:size
        count = 0

        neighbors = outneighbors(G, i)
        for j in 1:length(neighbors)
            if neighbors[j] == i
                continue
            else
                for k in j+1:length(neighbors)
                    if( !has_edge(G, j, k) )
                        count += 1
                    end
                end
            end
        end

        index = (min_count > count ) ? i : index
        min_count = (min_count > count ) ? count : min_count
    end

    return index
end

function _elimination_step(G::Graph, index::Int64)
    neighbors = outneighbors(G, index)
    for j in 1:length(neighbors)
        if neighbors[j] == index
            continue
        else
            for k in j+1:length(neighbors)
                if( !has_edge(G, j, k) )
                    add_edge!(G, j, k)
                end
            end
        end
    end
    rem_vertex!(G, index)

    return G
```

```
end

function _cholesky_concat(M::SparseMatrixCSC{Ban, Int64}, type::String = "minimum", min
::Bool = true )
    m = M.m
    n = M.n

    L = spzeros(Ban, n, m)
    for i in 1:n
        if( min && i != n )
            # calcolo l'indice di grado minimo
            if (cmp(type, "local") == 0)
                index = _minimum_local_degree(Graph(M[i:n,i:n]), M.n - i + 1)
            else
                index = _minimum_degree(Graph(M[i:n,i:n]))
            end
            # permuto riga e colonna per selezionare il pivot
            p = [v for v in 1:M.n-i+1]
            p[1] = index
            p[index] = 1
            M[i:n, i:n] = permute(M[i:n, i:n], p, p)
        end
        # ho portato il pivot in posizione (i, i)
        L[i, i] = sqrt( M[i, i] ) # pivot element

        for j in i+1:m
            L[j,i] = M[j, i]/L[i,i]
            for k in i+1:j
                M[j,k] = M[j,k] - (L[j,i]*L[k,i])
            end
        end
    end

    return L
end

function _cholesky!(M::AbstractMatrix, perm::Vector ) where { Tv<:AbstractAlgNum, Ti<:I
nteger }
    n, m = size(M)
    M = permute(M, perm, perm)
    T = typeof(M[1,1])
    L = spzeros(T, n, m)

    for i in 1:n
        L[i, i] = sqrt( norm(M[i, i]) )
        for j in i+1:m
            L[j,i] = M[j, i]/L[i,i]
            for k in i+1:j
                M[j,k] = M[j,k] - (L[j,i]*L[k,i])
            end
        end
    end

    return L
end


function _cholesky!(A::AbstractMatrix{T}, ::LinearAlgebra.Val{false}=LinearAlgebra.Val(
false); check::Bool = true) where T<:AbstractAlgNum
    LinearAlgebra.checksquare(A)
    if !LinearAlgebra.ishermitian(A);
```

```
        check && checkpositivedefinite(-1)

        return Cholesky(A, 'U', convert(LinearAlgebra.BlasInt, -1))
    else

        if( isa(A, SparseMatrixCSC) )
            P = amd(A)
            L = _cholesky!(A, P)
            return BanFactor(L, L', P)
        end

        isa(A, Hermitian) && (A = convert(Matrix{T}, A))
        C, info = LinearAlgebra._chol!(A, UpperTriangular);
        check && LinearAlgebra.checkpositivedefinite(info);

        return Cholesky(C.data, 'U', info)
    end
end

LinearAlgebra.cholesky!(A::AbstractMatrix{T}, ::LinearAlgebra.Val{false}=LinearAlgebra.
Val(false); check::Bool = true) where T<:AbstractAlgNum = _cholesky!(copy(A), LinearAlg
ebra.Val(false), check=check)
```

```
Entropy pool not available to seed RNG; using ad-hoc entropy sources.
```

Create Random Matrix with predefined sparsity

A necessary task we had to perform for the testing of our code consisted in the creation of a random,definite positive matrix with a certain level of sparsity.
The function will take as input the desired size,sparsity of the final matrix, and the probability of each entry of a cell to be nonzero.
The function will cycle, each time with a smaller probability of having a nonzero entry before eventually resetting, until we obtain a matrix with the desired sparsity. At this point the entries of the matrix will be converted to Ban, and the entire matrix returned in a sparse format.

In [5]:

```
function get_matrix(size::Int64, sparsity::Float64, prob::Float64 = (1-sparsity), BanTy
pe::Bool=true, )
    min_sparsity = (size*size - size) / size*size
    sparsity = (sparsity > min_sparsity ) ? min_sparsity : sparsity
    MAX_TRIES = 10

    A = sprand(size, size, prob )

    while(true)
        A = A'*A
        A = (A+A')/2

        nz = count(!iszero, A)
        s = 1 - nz/(size*size)
        v = eigvals(Matrix(A))

        if( s <= (sparsity + 0.2) && s >= sparsity  )
            A = A + I*( norm(v[1]) + 1 )
            break
        elseif( s < sparsity )
            prob = ( prob < 0.01 ) ? 0.01 : prob*0.90

            if( prob == 0.01 )
                MAX_TRIES -= 1
            end

            if( MAX_TRIES <= 0)
                prob = 0.9
                MAX_TRIES = 10
            end
        end

        A = sprand(size, size, prob )
    end

    if(BanType)
        A = A*one(Ban)
    end

    A = Symmetric(A)
    A = sparse(A)
    A = dropzeros(A)
    return A

end
```

Out[5]:

```
get_matrix (generic function with 3 methods)
```

In [7]:

```
M = get_matrix(3,0.5)
F = cholesky!(M)
println("Original Matrix:",M)
if(F.flag==1)
    invp = invperm(F.P)
    println("Reconstructed Matrix:",permute(F.L*F.Lt,invp,invp))
else
    println("Not Cholesky decomposition")
end
```

```
Original Matrix:
 α^0(1.0 + 0.0η^1 + 0.0η^2)                         ⋅
⋅
               ⋅                  α^0(1.000027778161792 + 0.0η^1 + 0.0η^2)  α
^0(0.004620030038336988 + 0.0η^1 + 0.0η^2)
               ⋅                α^0(0.004620030038336988 + 0.0η^1 + 0.0η^2)
α^0(2.57285675103868 + 0.0η^1 + 0.0η^2)
Reconstructed Matrix:
 α^0(1.0 + 0.0η^1 + 0.0η^2)                         ⋅
⋅
               ⋅                 α^0(1.0000277781617921 + 0.0η^1 + 0.0η^2)  α
^0(0.004620030038336988 + 0.0η^1 + 0.0η^2)
               ⋅                α^0(0.004620030038336988 + 0.0η^1 + 0.0η^2)
α^0(2.57285675103868 + 0.0η^1 + 0.0η^2)
```

LDL^T Decomposition

As stated before, the existance and uniqueness of the Cholesky decomposition lies in the hypotesis the matrix to which is applied is positive definitive.
If it's not, we can't perform a Cholesky decomposition.
However, if the matrix is symmetric and semi-positive definitive we can perform a LDL^T factorization:

$$A = LDL^T$$

where L is a lower triangular matrix with ones on the diagonal, D is a diagonal matrix and L^T is the transpose of L, therefore a upper triangular matrix.

The linear system $Ax = b$ can therefore be rewritten as follows:

$$LDL^T x = b$$

And, if we define $L^T x = z$,$Dz = y$ the system to be solved becomes:

$$\begin{cases} Ly = b \\ Dz = y \\ L^T x = z \end{cases}$$

.

A big advantage of the LDL^T in practice, compared to the Cholesky factorization, is the fact that the decomposition elimates the need for square-root operation, an operation particularly heavy.
The hypotesis for the existance of the LDL^T is the semi-definite positivity of the matrix. In our code we are aware that the matrices we pass in input to LDL^T are always semi-definite positive (by construction) therefore we prefered to avoid the check of this condition.
In fact, implementing the check of this condition, since there isn't a function to compute the eigenvalues for a matrix of Bans, would have resulted in computational increase, not justified in this case since we know there is no need for it.

LDL^T Factorization, pseudo-code

In [ ]:

```
for j in 1:n
        summ = 0
        for k in 1:j-1
            summ += (L[j, k]*Lt[j, k]*D[k])
        end

        D[j] = M[j, j] - summ

        for i in j+1:n
            summ = 0
            for k in 1:j-1
                summ += ( L[i,k]*Lt[j,k]*D[k])
            end
            L[i, j] = ((M[i,j] - summ)/ D[j])
            Lt[i, j] = L[i, j]
        end
    end
```

LDL^T Implementation

In [13]:

```
function _ldlt!(M::AbstractMatrix{T}) where T <: AbstractAlgNum
    n = size(M)[1]
    L = spdiagm(ones(Ban, n))
    Lt = spdiagm(ones(Ban, n))
    D = zeros(Ban, n)

    for j in 1:n
        summ = 0
        for k in 1:j-1
            summ += (L[j, k]*Lt[j, k]*D[k])
        end

        D[j] = M[j, j] - summ

        for i in j+1:n
            summ = 0
            for k in 1:j-1
                summ += ( L[i,k]*Lt[j,k]*D[k])
            end
            L[i, j] = ((M[i,j] - summ)/ D[j])
            Lt[i, j] = L[i, j]
        end
    end

    return L, spdiagm(D), Lt'
end

function _ldlt!(A::AbstractMatrix{T}, perm::AbstractVector, isSparse::Bool = true ) whe
re T <: AbstractAlgNum
    M = permute(A, perm, perm)
    n = size(M)[1]
    L = spdiagm(ones(Ban, n))
    Lt = spdiagm(ones(Ban, n))
    D = zeros(Ban, n)

    for j in 1:n
        summ = 0
        for k in 1:j-1
            summ += (L[j, k]*Lt[j, k]*D[k])
        end

        D[j] = M[j, j] - summ

        for i in j+1:n
            summ = 0
            for k in 1:j-1
                summ += ( L[i,k]*Lt[j,k]*D[k])
            end
            L[i, j] = ((M[i,j] - summ)/ D[j])
            Lt[i, j] = L[i, j]
        end

    end

    return L, spdiagm(D), Lt'
end

function ldlt(M::AbstractMatrix{T}, sparsity_t::Float64 = 0.0, delta_t::Float64 = 0.5)
where T <: AbstractAlgNum
```

```
    A = copy(M)
    LinearAlgebra.checksquare(A)
    if !LinearAlgebra.ishermitian(A);
        return BanFactor()
    else
        if( sparsity_t >= delta_t )
            A = sparse(A)
            P = amd(A)
            L, D, Lt = _ldlt!(A, P)
            b = BanFactor(L, D, Lt, P)
            return b
        end
        L, D, Lt = _ldlt!(A)
        return BanFactor(L, D, Lt)
    end
end
```

Out[13]:

```
ldlt (generic function with 3 methods)
```

In [8]:

```
M = get_matrix(3,0.5)
F = ldlt(M)
println("Original Matrix:",M)
if(F.flag==0)
    println("Reconstructed Matrix:",F.L*F.D*F.Lt)
else
    println("Not Cholesky decomposition")
end
```

```
Original Matrix:
 α^0(1.0 + 0.0η^1 + 0.0η^2)                          ⋅
⋅
             ⋅                α^0(1.0591006705983121 + 0.0η^1 + 0.0η^2)  α^0
(0.1292578168419506 + 0.0η^1 + 0.0η^2)
             ⋅                α^0(0.1292578168419506 + 0.0η^1 + 0.0η^2)  α^0
(1.2826970158816502 + 0.0η^1 + 0.0η^2)
Reconstructed Matrix:
 α^0(1.0 + 0.0η^1 + 0.0η^2)                          ⋅
⋅
             ⋅                α^0(1.0591006705983121 + 0.0η^1 + 0.0η^2)  α^0
(0.1292578168419506 + 0.0η^1 + 0.0η^2)
             ⋅                α^0(0.1292578168419506 + 0.0η^1 + 0.0η^2)  α^0
(1.2826970158816502 + 0.0η^1 + 0.0η^2)
```

Performance Comparison

Benchmark for the comparison performance of original Cholesky,Cholesky for sparse,LDL^T for Sparse of Ban

In [2]:

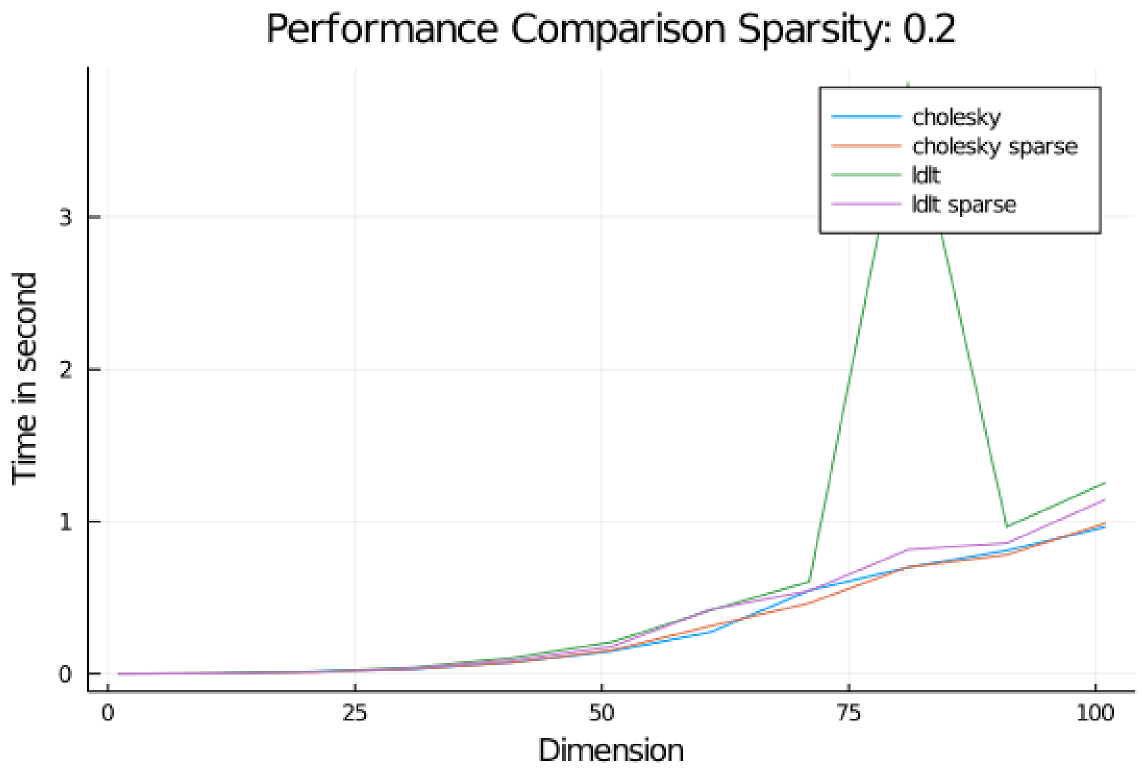
```
using DataFrames
using CSV
using Plots
using AMD
using BenchmarkTools
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

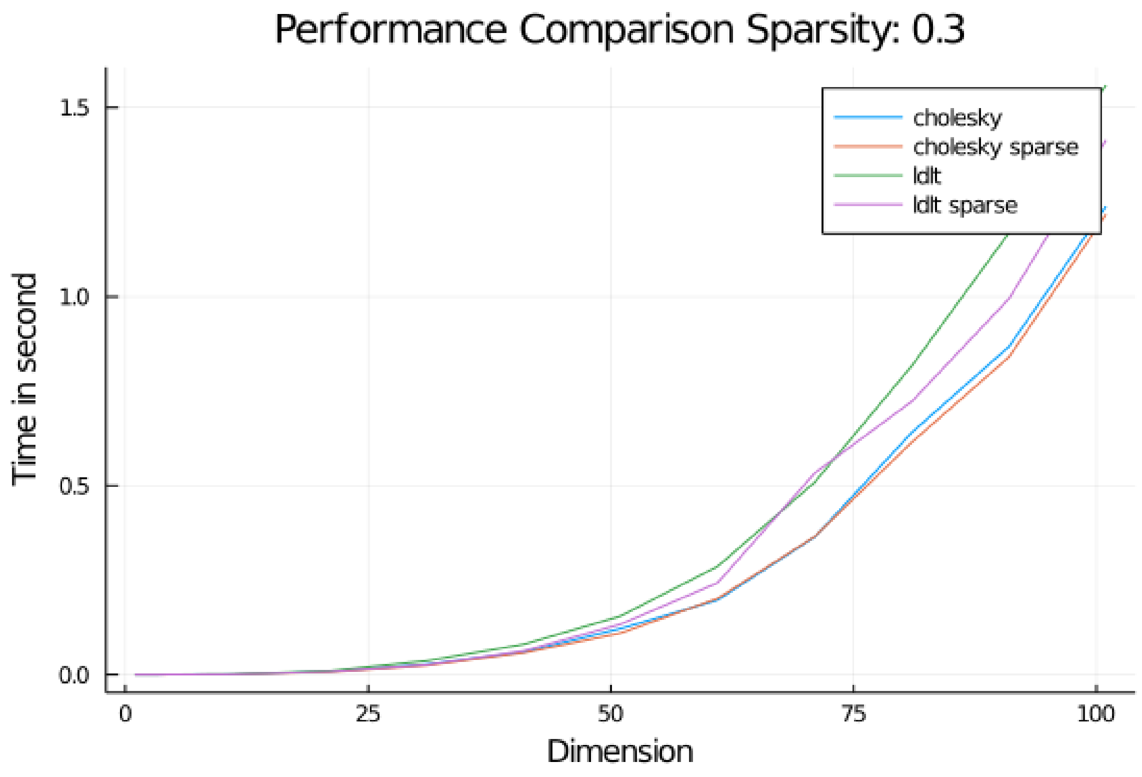
Out[2]:

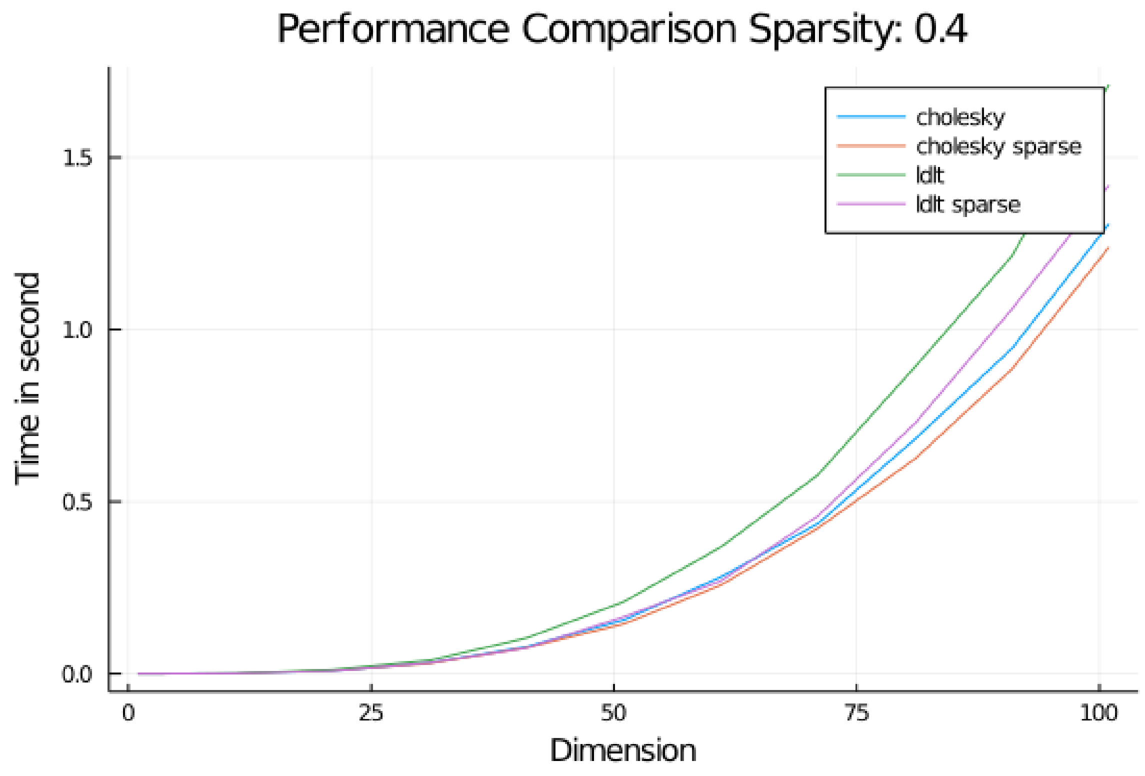
```
10
```

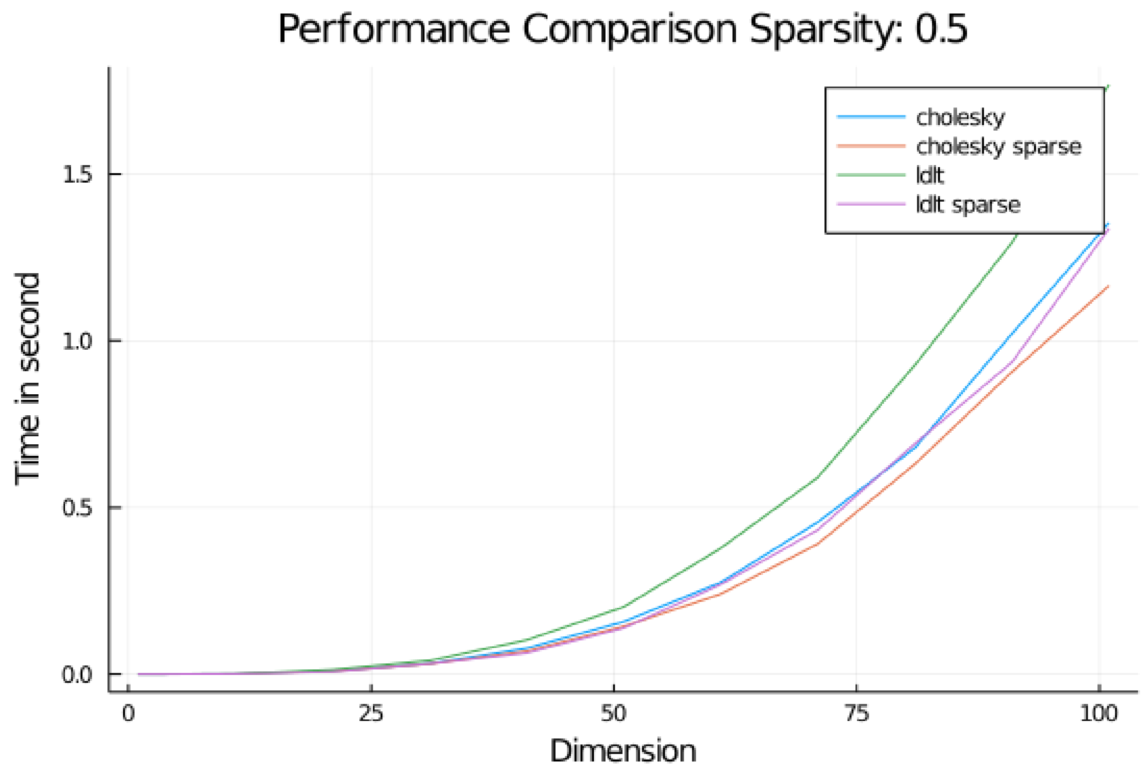
In [9]:

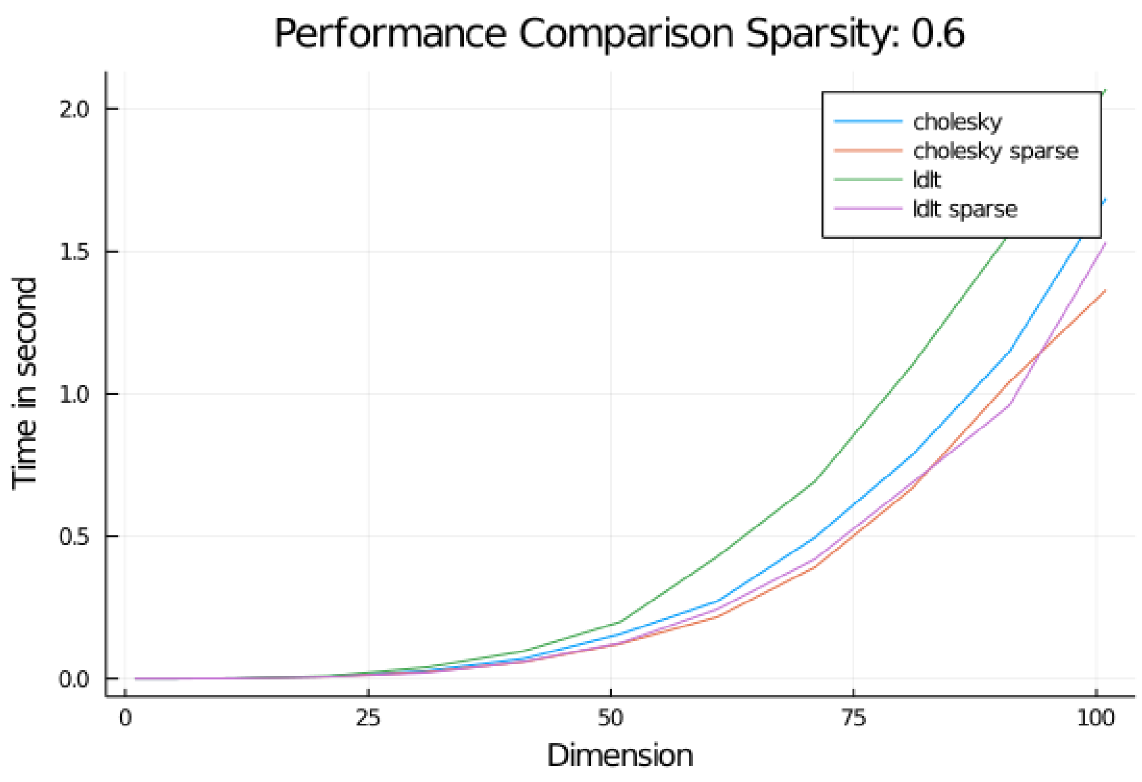
```
for s in range(0.2, 0.80, step = 0.1)
    xs = Float64[]
    ys1 = Float64[]
    ys2 = Float64[]
    ys3 = Float64[]
    ys4 = Float64[]
    println("")
    for n in range(1, 101, step=10)

        #println("N = ", n)
        push!(xs, n)
        #println("---> cholesky")

        b1 = @benchmark cholesky(Matrix(x))  setup = ( x = get_matrix($n, $s) )
        push!(ys1, mean(b1).time/10^9)
        b2 = @benchmark cholesky!(x)  setup = ( x = get_matrix($n, $s) )
        push!(ys2, mean(b2).time/10^9)
        #println("---> ldlt ")
        b3 = @benchmark ldlt(x)  setup = ( x= get_matrix($n, $s) )
        push!(ys3, mean(b3).time/10^9)
        b4 = @benchmark ldlt(x,0.99)  setup = ( x= get_matrix($n, $s) )
        push!(ys4, mean(b4).time/10^9)

        #print("\n")


        #println("DONE!")
        #println("")
    end

    Title = "Performance Comparison Sparsity: "*string(s)
    display(plot(xs, [ys1,ys2,ys3,ys4], label=["cholesky" "cholesky sparse" "ldlt" "ldl
t sparse"], xaxis="Dimension", yaxis="Time in second", title=Title, fmt = :png))
end
```

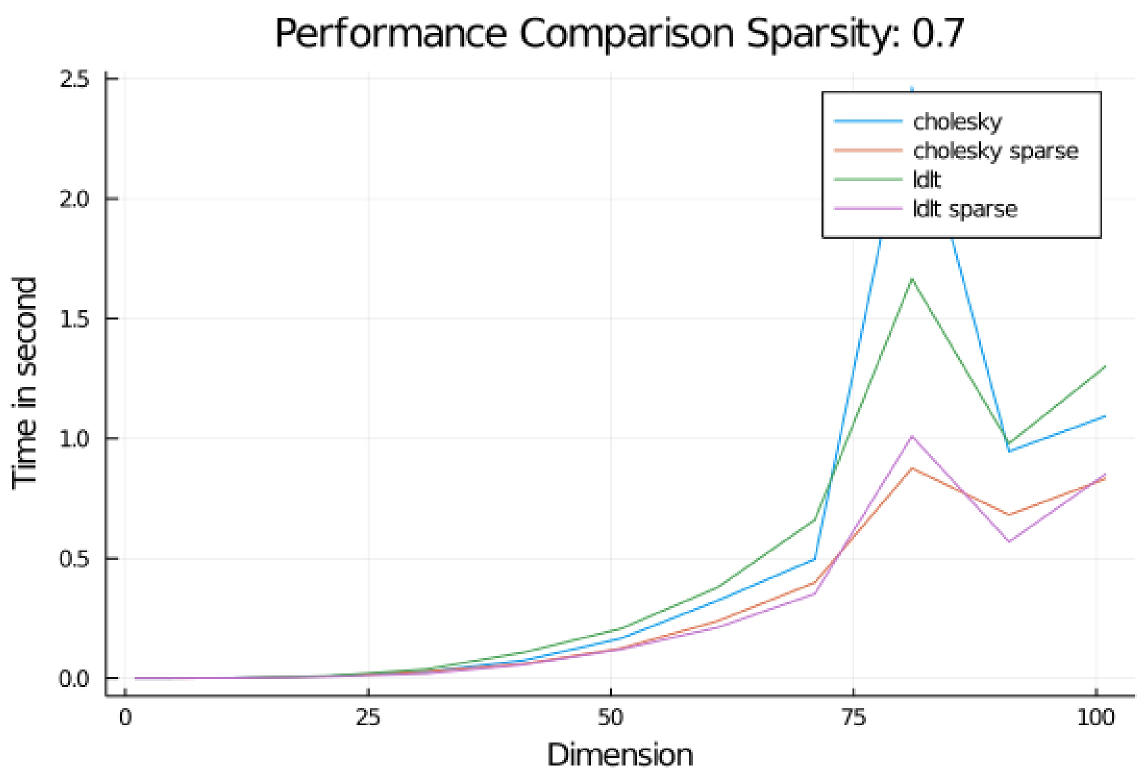

Performance Comparison Sparsity: 0.2
cholesky
cholesky sparse
ldlt
ldlt sparse
Time in second
0
1
2
3
0
25
50
75
100
Dimension

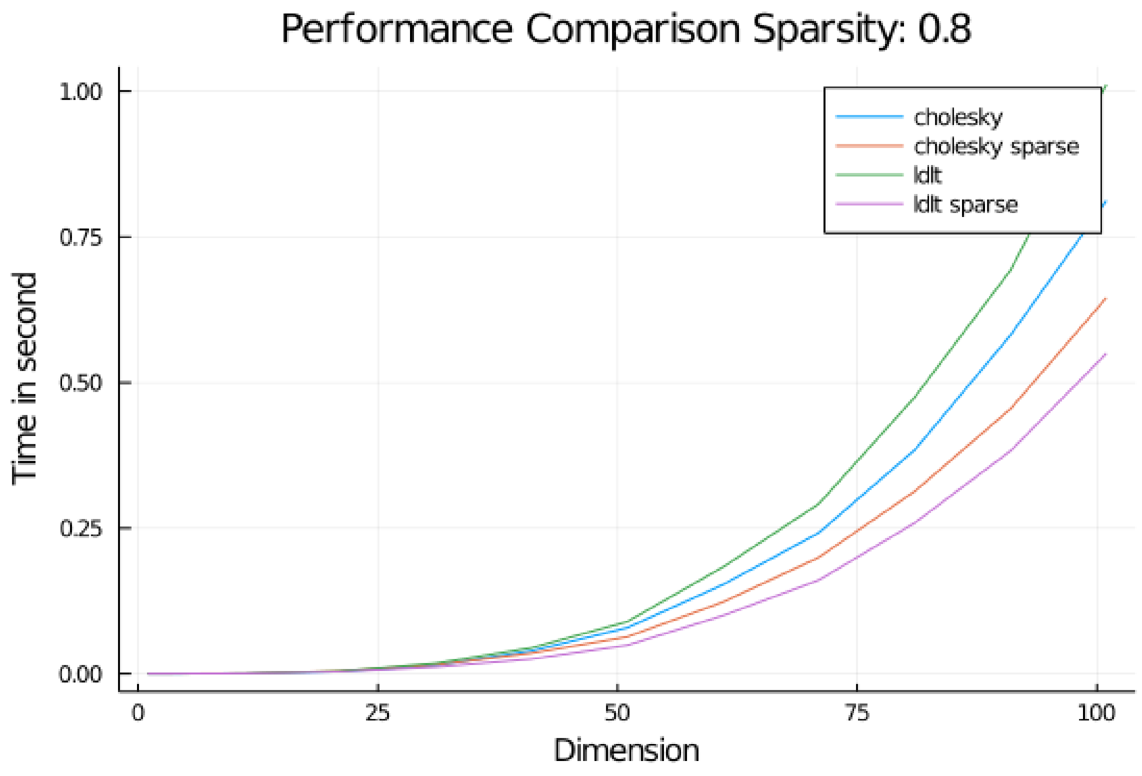

Performance Comparison Sparsity: 0.3
cholesky
cholesky sparse
ldlt
ldlt sparse
Time in second
0.0
0.5
1.0
1.5
0
25
50
75
100
Dimension


Performance Comparison Sparsity: 0.4
Time in second
Dimension
cholesky
cholesky sparse
ldlt
ldlt sparse
0.0
0.5
1.0
1.5
0
25
50
75
100


Performance Comparison Sparsity: 0.5
Time in second
Dimension
cholesky
cholesky sparse
ldlt
ldlt sparse
0.0
0.5
1.0
1.5
0
25
50
75
100


Performance Comparison Sparsity: 0.6
cholesky
cholesky sparse
ldlt
ldlt sparse
Time in second
Dimension
0.0
0.5
1.0
1.5
2.0
0
25
50
75
100


Performance Comparison Sparsity: 0.7
cholesky
cholesky sparse
ldlt
ldlt sparse
Time in second
Dimension
0.0
0.5
1.0
1.5
2.0
2.5
0
25
50
75
100



The performance of the various factorization varies depending from the sparsity level (we will use '>' to indicate better performance, meaning less execution time:

- For sparsity level equal to 0.1, for small dimensions all the decomposition have basically the same performance. As we increase the dimension, cholesky > cholesky sparse > ldlt sparse > ldlt</p>
- For sparsity level equal to 0.2, 0.3 and 0.4, for small dimension all the decomposition have basically the same performance. Since this is a behaviour common to the previous level of sparsity, and all the next ones, we will avoid to repeat this concept. However, as we expected, as we progress through the sparsity level the cholesky sparse will get better than the normaly cholesky. In this case in fact we have cholesky sparse > cholesky > ldlt sparse > ldlt</p>
- For sparsity level equal to 0.5 and 0.6, not only the cholesky sparse is still getting better and better compared to the cholesky, but the ldlt has also performance that results better than the cholesky. We will have the following performance ordering cholesky sparse > ldlt sparse > cholesky >ldlt</p>
- For sparsity level equal to 0.7 and 0.8, the ldlt has gotten so better through the sparsity increase that it is also able to overcome in term of performance even the cholesky sparse, that is now by far better than the cholesky. The performance are ldlt sparse > cholesky sparse >cholesky > ldlt</p>

Why LDL^T sparse is better than Cholesky sparse?

To investigate further the reasons why the LDL^T is able, for high level of sparsity, to obtain better performance than the Cholesky sparse, we perform various tries in order to find out if there were some operation particularly computationally heavy.
In particular we analyzed, in the case of dim=100:

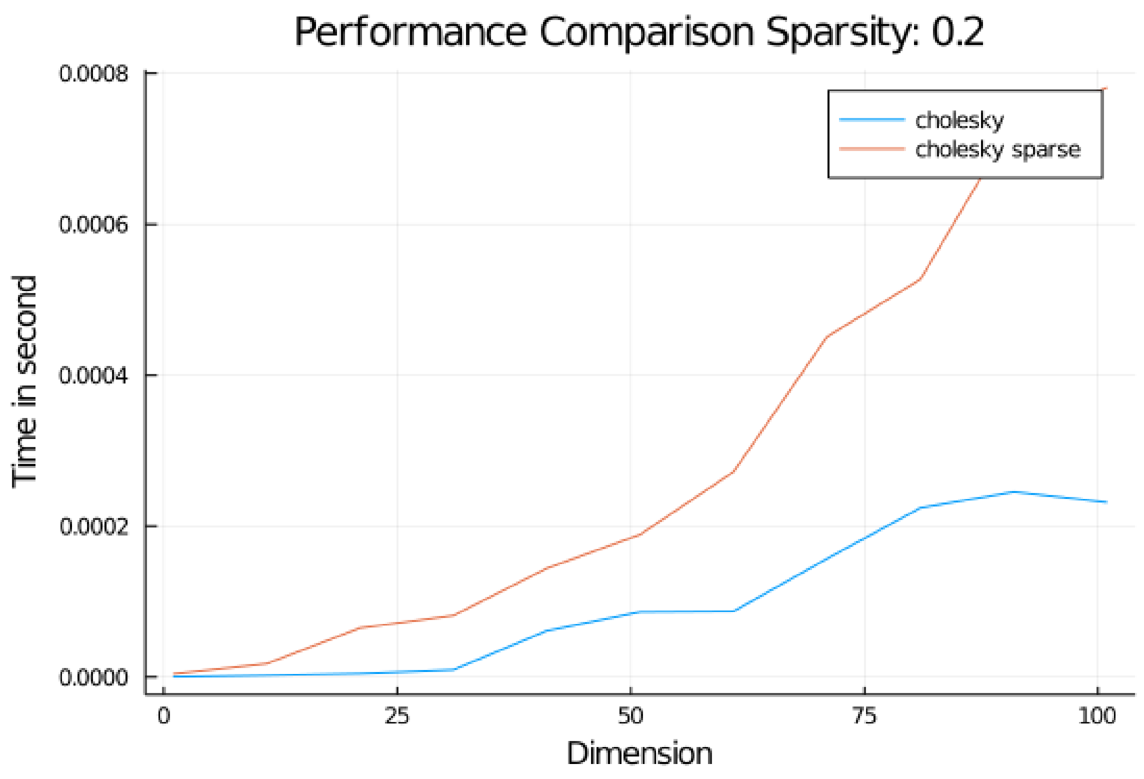
- The time of execute all the square roots of the diagonal elements</p>
- The additional checks performed</p>
- The time to execute the single operations</p>

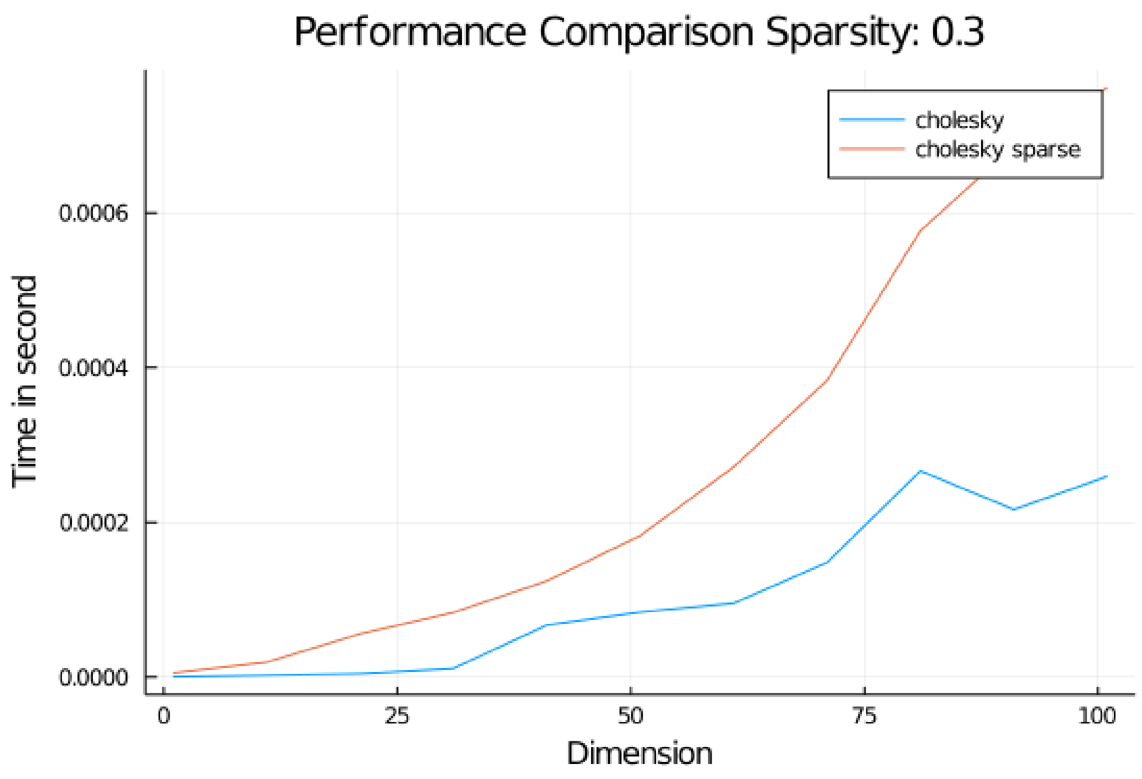
We weren't able to find decisive proofs on why this really happens, since none of the investigations we performed results in really time consuming operations.
Searching through the papers on the argument, we have however found out that this is a well known behaviour: in this paper (https://www.researchgate.net/publication/227126109_A_fast_LDL-factorization_approach_for_large_sparse_positive_definite_system_and_its_application_to_one-to-one_marketing_optimization_computation) a similar trend on the performance is verified.

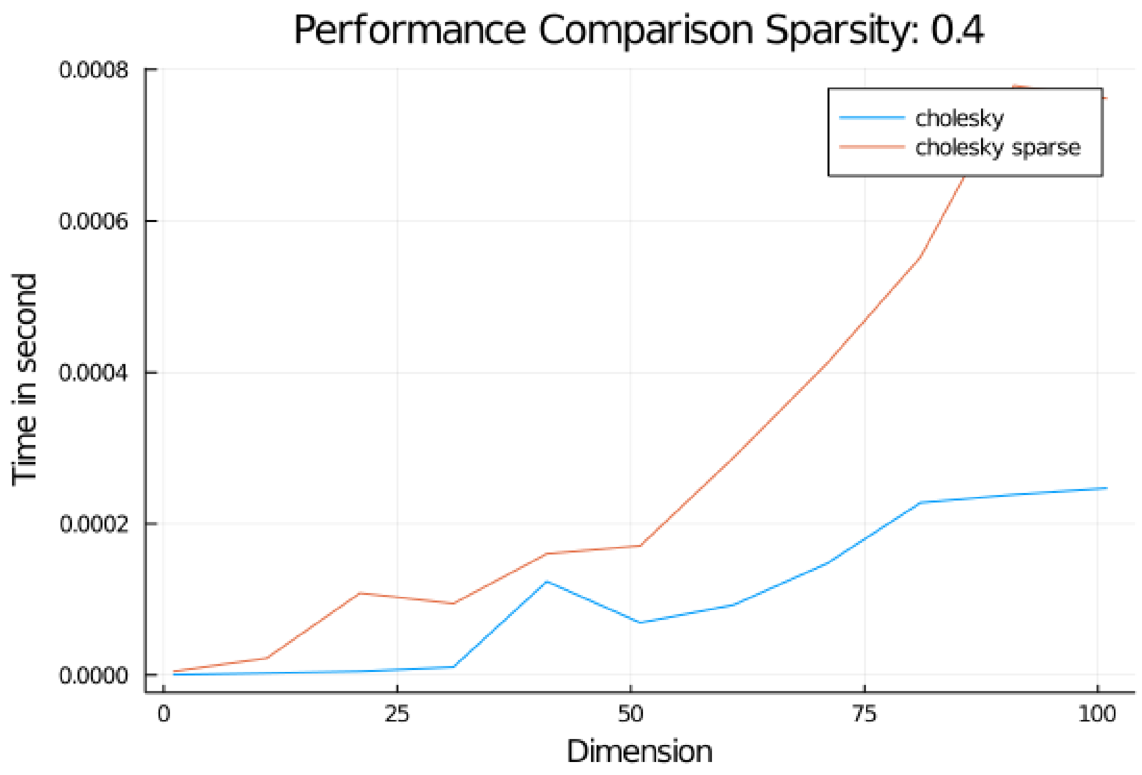
Benchmark for the comparison performance of original Cholesky,Cholesky for sparse, for Float64

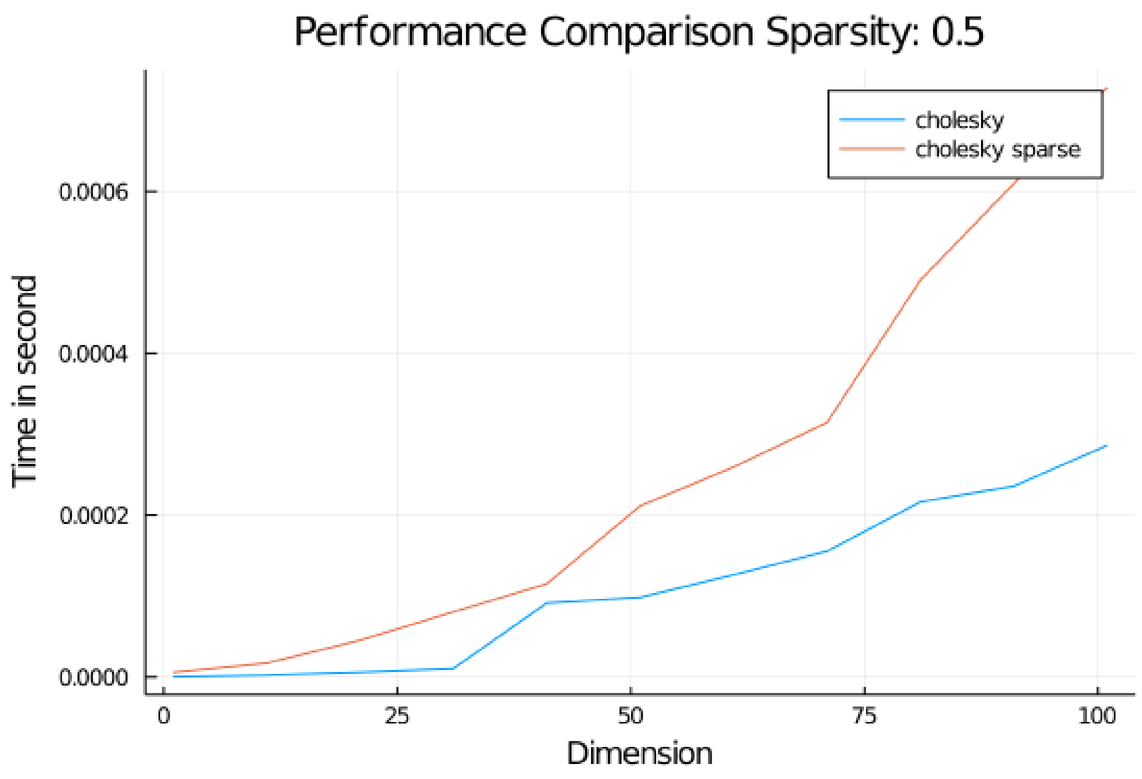
In [13]:

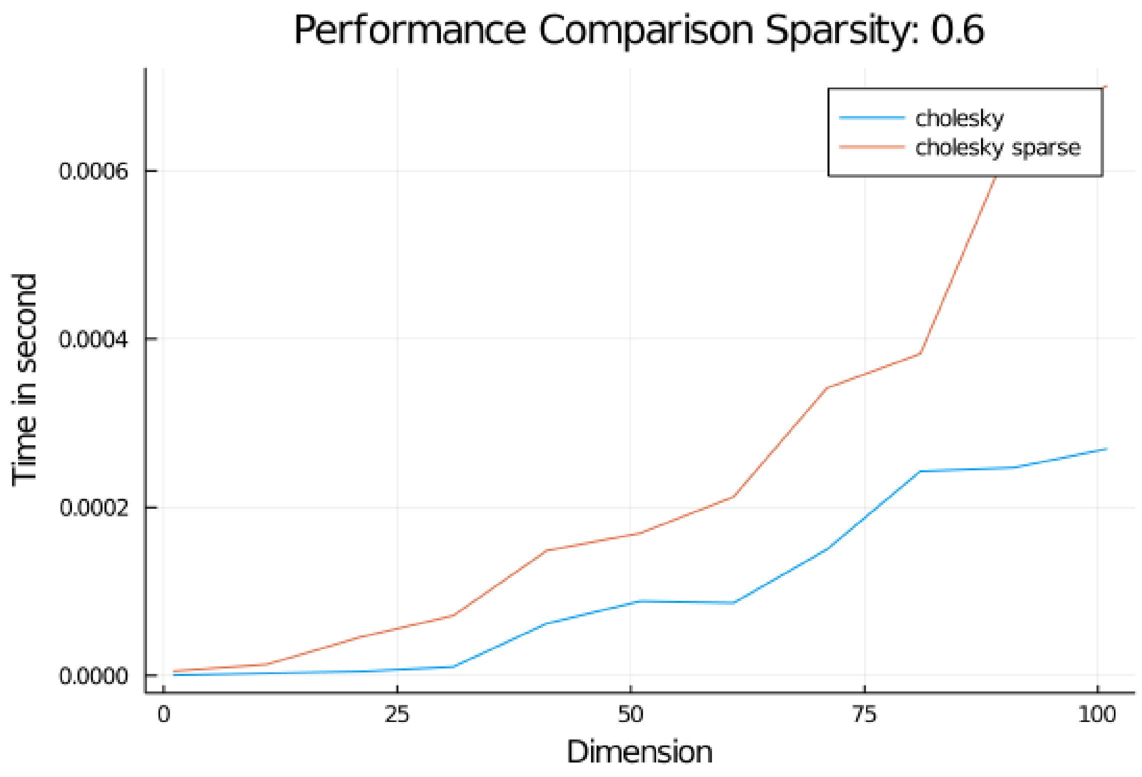
```
for s in range(0.2, 0.80, step = 0.1)
    xs = Float64[]
    ys1 = Float64[]
    ys2 = Float64[]
    println("")
    for n in range(1, 101, step=10)

        #println("N = ", n)
        push!(xs, n)
        #println("---> cholesky")

        b1 = @benchmark cholesky(Matrix(x))  setup = ( x = get_matrix($n, $s,1-$s,false
) )
        push!(ys1, mean(b1).time/10^9)
        b2 = @benchmark cholesky(x)  setup = ( x = get_matrix($n, $s,1-$s,false) )
        push!(ys2, mean(b2).time/10^9)
    end

    Title = "Performance Comparison Sparsity: "*string(s)
    display(plot(xs, [ys1,ys2], label=["cholesky" "cholesky sparse"], xaxis="Dimension"
, yaxis="Time in second", title=Title, fmt = :png))
end
```

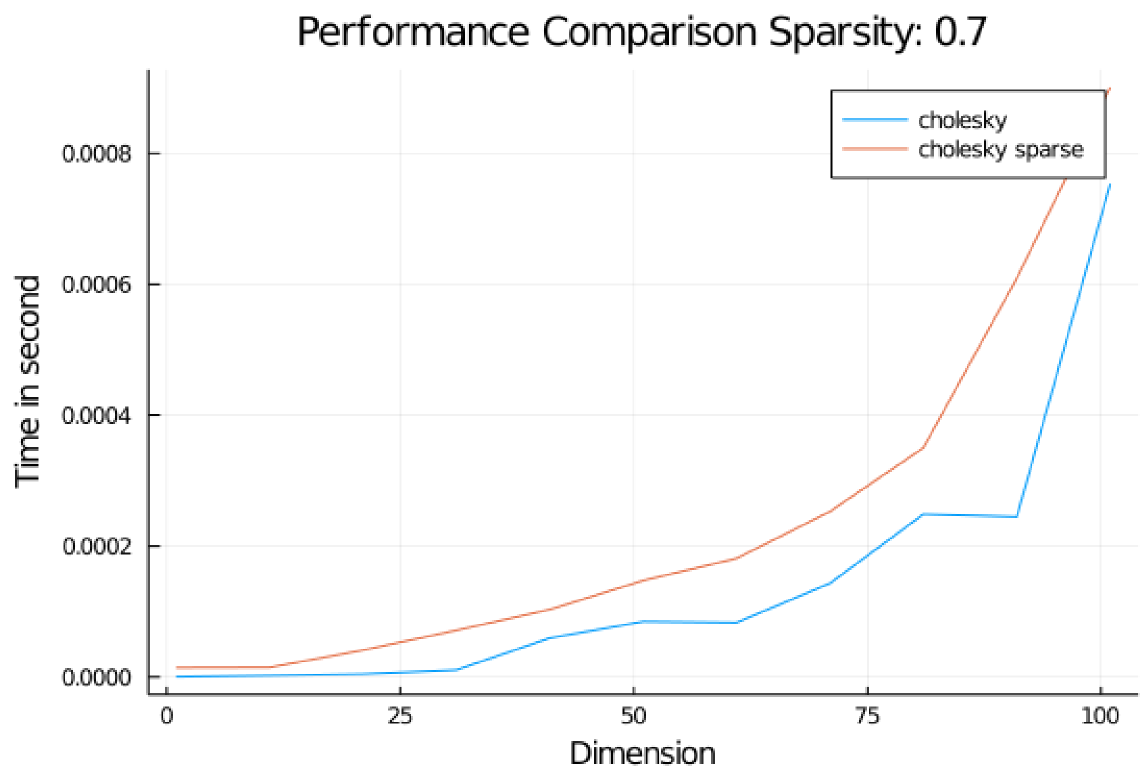

Performance Comparison Sparsity: 0.2
Time in second
0.0008
0.0006
0.0004
0.0002
0.0000
0
25
50
75
100
Dimension
cholesky
cholesky sparse

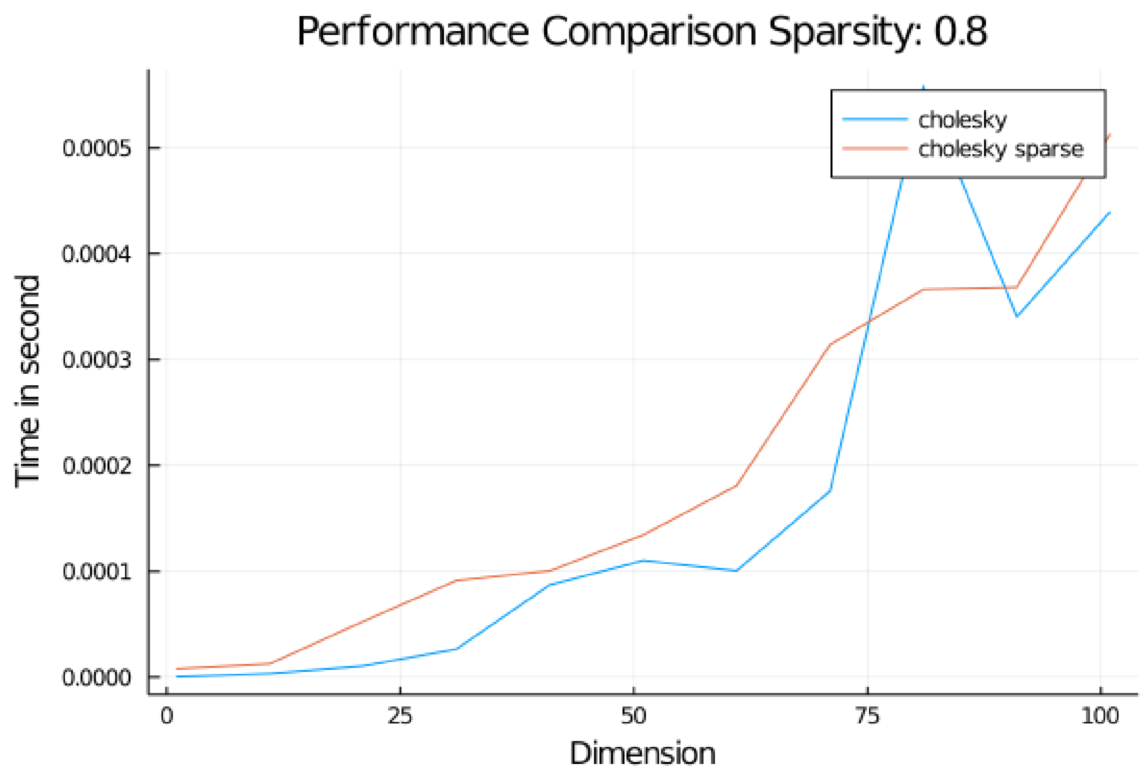

Performance Comparison Sparsity: 0.3
cholesky
cholesky sparse
Time in second
0.0000
0.0002
0.0004
0.0006
0
25
50
75
100
Dimension


Performance Comparison Sparsity: 0.4
cholesky
cholesky sparse
Time in second
0.0000
0.0002
0.0004
0.0006
0.0008
0
25
50
75
100
Dimension


Performance Comparison Sparsity: 0.5
cholesky
cholesky sparse
Time in second
0.0006
0.0004
0.0002
0.0000
0
25
50
75
100
Dimension


Performance Comparison Sparsity: 0.6
cholesky
cholesky sparse
Time in second
0.0006
0.0004
0.0002
0.0000
0
25
50
75
100
Dimension


Performance Comparison Sparsity: 0.7
cholesky
cholesky sparse
Time in second
0.0000
0.0002
0.0004
0.0006
0.0008
Dimension
0
25
50
75
100


Performance Comparison Sparsity: 0.8
cholesky
cholesky sparse
Time in second
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
Dimension
0
25
50
75
100

In the case of the Float64 type, the factorization through the Cholesky is always better than the factorization through the Cholesky sparse.
However, this is expected. Since in the case of Float64 we are making use of graphic accelerator to speed up the computation (and this can be notice by how fast the execution times are), the advantage we gain by performing the minimum degree ordering, and therefore avoiding some computation, it is not justified since how fast each single operation is to be executed.

Quadratic System resolution

From the teory, we know the unreduced form of the quadratic system is:

$$\begin{bmatrix} 0 & A & 0 \\ A^T & -Q & I \\ 0 & S & X \end{bmatrix} \begin{bmatrix} \Delta\lambda \\ \Delta x \\ \Delta s \end{bmatrix} = \begin{bmatrix} -r_b \\ -r_c \\ -r_{xs} \end{bmatrix}$$

The system can be made explicit (in the program the r_c,r_{xs},r_b are passed with inverted sign, we will follow this assumption) :

$$A^T\Delta\lambda - Q\Delta X + I\Delta s = r_c$$
$$S\Delta x + X\Delta s = r_{xs}$$
$$A\Delta x = r_b$$

From the second equation

$$\Delta s = X^{-1}(r_{xs} - S\Delta x)$$

$$A^T\Delta\lambda - Q\Delta X - X^{-1}S\Delta x = r_c - X^{-1}r_{xs}$$

$$D = Q + X^{-1}S$$

$$A^T\Delta\lambda - D\Delta X = r_c - X^{-1}r_{xs}$$

$$D^{-1}A^T\Delta\lambda - I\Delta X = D^{-1}(r_c - X^{-1}r_{xs})$$

From the third equation $A\ \Delta x$ = r_b , also we premultiply everything for A

$$AD^{-1}A^T\Delta\lambda - r_b = AD^{-1}(r_c - X^{-1}r_{xs})$$

$$AD^{-1}A^T\Delta\lambda = r_b + AD^{-1}(r_c - X^{-1}r_{xs})$$

$$b = r_b + AD^{-1}(r_c - X^{-1}r_{xs})$$

$$f_1 = AD^{-1}A^T$$

$$\Delta\lambda = f_1 \setminus b$$

To retrieve the cascading terms:

$$A^T\Delta\lambda - Q\Delta X + \Delta s = r_c$$

$$-Q\Delta X + \Delta s = r_c - A^T\Delta\lambda$$

$$\Delta s = r_c - A^T\Delta\lambda + Q\Delta x$$

$$S\Delta x + X\Delta s = r_{xs}$$

$$S\Delta x + X(r_c - A^T\Delta\lambda + Q\Delta X) = r_{xs}$$

$$\Delta x(S + QX) = r_{xs} + Xr_c + XA^T\Delta\lambda$$

$$S_1 = S + (XQ)$$

$$b_1 = r_{xs} + Xr_c + XA^T\Delta\lambda$$

$$\Delta x = S_1 \setminus b_1$$

$$A^T\Delta\lambda - Q\Delta X + \Delta s = r_c$$

$$\Delta s = r_c - A^T\Delta\lambda + Q\Delta x$$

$$\begin{bmatrix} -Q & I \\ S & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \end{bmatrix} = \begin{bmatrix} r_c - A^T\Delta\lambda \\ r_{xs} \end{bmatrix}$$

In [ ]:

```
# unreduced form
function check_diagonal(_Q, num_var)
    Q1 = _Q[1:num_var, 1:num_var]

    for i in 1:num_var
        for j in 1:(i-1)
            if( Q1[i, j] != 0 )
                return false
            end
        end
    end

    return true
end

function fact4(A,Q,x,s, tol=1e-8, Q_diag = false)
#    println("Actual point: ")
#    println("x: ", x)
#    println("s: ", s)

    # PROBLEMA :
    # IL DENOISING MI MANDA A ZERO LE VARIABILI SOTTO LA  TOLLERANZA!

    n, m = size(A)
    num_var = m - n
    #S = [(si == 0) ? tol : si for si in s]
    #X = [(xi == 0) ? tol : xi  for xi in x ]

    S = copy(s)
    X = copy(x)

    # OSS:
    #      D = Q + invX*S
    #      D è diagonale a blocchi
    #      perchè Q è diagonale a blocchi e invX*S è diagonale
    #      In particolare D sarà così formata
    #      (Q + I*invX*S)[1:num_var, 1:num_var]        0
    #          0                                  I*invX*S[1:num_var, 1:num_var]
    #
    #      quindi quando andiamo a calcolare l'inversa
    #      inv(Q + I*invX*S)[1:num_var, 1:num_var]        0
    #          0                                   inv(I*invX*S[1:num_var, 1:num_va
r])
    #
    #      ovvero
    #      inv(Q + I*invX*S)[1:num_var, 1:num_var]        0
--->     M1   0
    #          0                                   I*invS*X[num_var+1:n , num_var+1:]
0   M2
    #

    # Quindi  mi servono solo le prime num_var componenti dell'inversa di X e le restan
ti dell'inversa di S
    n_x = length(x)
    invX = [1/X[i]  for i in 1:num_var ]
    invS = [1/S[i]  for i in num_var+1:m ]

    M1 = []
    if( Q_diag )
```

```
        dQ = Q[diagind(Q[1:num_var, 1:num_var])] + (invX.*S[1:num_var])
        M1 = diagm( [1/dqi for dqi in dQ ])
    else
        M1 = inv( Q[1:num_var, 1:num_var] + diagm( invX.*S[1:num_var]  ) )
    end

    M2 = diagm( invS.* X[num_var+1:m]  )

    invD = [          M1             zeros(num_var, m - num_var);
            zeros(m-num_var, num_var)            M2                ]


    AA = A*invD*A'
    AA = Symmetric(AA)

    f1 = []
    AA = sparse(AA)
    nA,mA = size(AA)
    sp = 1 - (length(AA.nzval)/(nA*mA))
    if(sp>=0.7)
        f1 = BAN.ldlt( AA, sp )
    else
        try
            f1 = cholesky!( sparse(AA) )
        catch PosDefException
            f1 = BAN.ldlt( AA, sp )
        end
    end

    if( f1.flag == UInt8(2) )
        println("fattorizzazione fallita")
        exit(0)
    end

    M = [   -Q  I   ;
            diagm(S)   diagm(x)   ]

   f2 = lu(M)

    return f1, f2, invD
end


function solve4(f1,f2, A, Q, invD, x, s, rb, rc, rxs, tol = 1e-8)
    invX = [(xi == 0) ? 1/tol : 1/xi  for xi in x ]

    b = rb +(A*invD*rc) - (A*invD*diagm(invX)*rxs)

    dlam = f1\b

    c = rc - A'*dlam
    sol = f2\[c; rxs]
    n = length(rc)

    dx = sol[1:n]
    ds = sol[1+n:2*n]

    return dlam, dx, ds
end
```

Analysis of the matrix D and $AD^{-1}A^T$

From the resolution of the quadratic system:

$$D = Q + X^{-1}S$$

The matrix D composed by three matrices:

- Diagonalized of s
- The inverse matrix of diagonalized x
- The matrix Q

Both the diagonalized of S and X are definite positive, since the Interior Point Method says that such values are positive, otherwise we would be on the border of the polyedrom. The product of this two diagonal matrices is still a diagonal matrix. Q is also a matrix positive definite,symmetric and block diagonal.
The matrix D, sum of two diagonal matrices, is therefore diagonal (and given it's diagonal it's also symmetric) and definite positive.

$$AA = AD^-1A$$

The AA matrix is the product of a diagonal matrix,inverse of D, and the matrix A and it's transposed. The symmetry is preserved, and in general the matrix AA is semi-definite positive.

However, the symmetry of the matrix is AA is preserved only in theory. In practice, when we compute this matrix(in the case the matrix is made of Bans) some noise artifacts appeared on the components lesser than the tolerance.
We resolved this problem by applying the Symmetric(.) operator; this operator will take the values lesser than the tolerance and force them to be equal.

How to compute the inverse of the matrix D

In julia the inversion of a matrix is performed through a LU factorization; in our case, since we have focused on how to improve the performance through the application of the either Cholesky or LDL^T factorization, it wouldn't have made sense to keep this approach. Therefore we decided to do the inverse of the matrix D manually.
As previously stated, the matrix D, necessary to compute

$$\Delta\lambda = f_1 \setminus b$$

where $f_1 = AD^{-1}A^T$ and $b = r_b + AD^{-1}(r_c - X^{-1}r_{xs})$
is equal to

$$Q + X^{-1}S$$

.
We noticed that, in reality, D is a matrix formed by two blocks:

$$\begin{bmatrix} Q + X^{-1}S[1:numVar, 1:numVar] & 0 \\ 0 & X^{-1}S[numVar+1:dim, numVar+1:dim] \end{bmatrix}$$

We have one block, on the top left, which diagonality depends from the diagonality of the Q, because it's Q plus a diagonal block. In this case Q it's not the Q of the standard problem, but the Q of the original problem, the real variables without the slack variables.
On the bottom right we have the other block, the partion of Q of the slack variables, but that part is all equal to 0, plus the remaining part of the diagonal matrix $X^{-1}S$
If we apply the inverse operator to Q we obtain:

$$\begin{bmatrix} inv(Q + X^{-1}S[1:numVar, 1:numVar]) & 0 \\ 0 & S^{-1}X[numVar+1:dim, numVar+1:dim] \end{bmatrix}$$

All we need to obtain are the first numVar components of X^{-1} and the remaining component,from numVar+1 to dim, of S
If we define $M1 = inv(Q + X^{-1}S[1:numVar, 1:numVar])$ and $M2 = S^{-1}X$, the matrix Q become.

$$\begin{bmatrix} M1 & 0 \\ 0 & M2 \end{bmatrix}$$

The matrix $M2$ can be easily computed through the product of the last components of S^{-1} and X.
On the other hand, we can't compute $M1$ manually but we need to use the inv(.). That is, unless the matrix Q is diagonal itself, in which case I can manually inverte the elements of
$inv(Q + X^{-1}S[1:numVar, 1:numVar]$

In [ ]:

```
invX = [1/X[i]  for i in 1:num_var ]
invS = [1/S[i]  for i in num_var+1:m ]

M1 = []
if( Q_diag )
    dQ = Q[diagind(Q[1:num_var, 1:num_var])] + (invX.*S[1:num_var])
    M1 = diagm( [1/dqi for dqi in dQ ])
else
    M1 = inv( Q[1:num_var, 1:num_var] + diagm( invX.*S[1:num_var]  ) )
end

M2 = diagm( invS.* X[num_var+1:m]  )

invD = [          M1              zeros(num_var, m - num_var);
          zeros(m-num_var, num_var)            M2                ]
```

When is Q diagonal?

The check to verify if Q is a diagonal matrix can be done just one time, because the matrix won't be modified through the progression of the Interior Point Method.
In addition, Q is symmetric and the control for the diagonality can be restricted only to the block on top left, which is the block of the coefficients relative to the real variables. All the rest of the matrix will be 0.
Since it's symmetric, all we have to do a search on the inferior tridiagonal of this block. If we define as n the number of real variables, the complexity of this search is equal to $O(n^2/2)$

In [ ]:

```
function check_diagonal(_Q, num_var)
    Q1 = _Q[1:num_var, 1:num_var]

    for i in 1:num_var
        for j in 1:(i-1)
            if( Q1[i, j] != 0 )
                return false
            end
        end
    end

    return true
end
```

Denoising phenomenon

In the Interior Point Method we move through the center of the feasible region towards the optimum, therefore the variable X(primal variables) and S(slack variables) are > 0.
Since X and S are strictly positive, the product $X^{-1}S$ exists and it's definite positive.
However, as we get closer and closer to a constraint, some xi and some si will go towards the 0, but they will never be 0.
This is a problem with the Denoising operation. The idea of the IPQP is to first optimize the first monosemia, the most important one; once we have achieved the convergence on the first mosemia, we apply the denoising operation, which works by setting the small monosemia coefficients of X and S to 0 and update the discrepancy vector. This means,practically, that since we have converged on the leadomg monosemia we are not interested anymore into progressing in its optimization, and therefore we continue by optimizing the first monosemia.
The drawback of it is the fact that some xi and si are now equal to 0. But this means that there is no more guarantee to the definite positive of X and S, and also makes impossible to compute their inverse.
The first solution we decided to adopt was, in the case the denoising operation has taken a xi or si equal to 0, to reassign them to a value equal or less to the tolerance.

In [ ]:

```
S = [(si == 0) ? tol : si for si in s]
X = [(xi == 0) ? tol : xi  for xi in x ]
```

However this caused instabilities during the optimization of lexicographic problems: if we add a finite number, that number will be infinitely larger than η that we are going to optimize on the other objects.
This issue was resolve by modifying the value the si and xi equal to 0 are going to take: instead of taking the constant tol, we assign them to a η smaller than the ones we are actually minimizing; in this manner,minimizing this value will be infinitely less important than minimizing the others.
Our system will avoid to minimize that variable, while also avoiding to have inconsistencies in the matrix.

In [ ]:

```
noise = 100*tol*η^(n_levels)

s_a = [(s[i] == 0) ? noise : s[i] for i in 1:length(s)]
x_a = [(x[i] == 0) ? noise : x[i] for i in 1:length(x) ]
```

Factorizing f1, practically

In [ ]:

```
AA = sparse(AA)
nA,mA = size(AA)
sp = 1 - (length(AA.nzval)/(nA*mA))
if(sp>=0.7)
    f1 = BAN.ldlt( AA, sp )
else
    try
        f1 = cholesky!( sparse(AA) )
    catch PosDefException
        f1 = BAN.ldlt( AA, sp )
    end
end
```

When we factorize f1, we do a try catch:

- First, we control the sparsity of the matrix AA. As we found out from our benchmarks, it is convenient to use LDL^T instead of Cholesky for sparsity >=0.7</p>
- If the sparsity of the matrix is greater or equal than 0.7, we factorize using LDL^T </p>
- If it's not we perform a try-catch</p>
- In the try we control if we can perform a cholesky factorization. If it's not possible,the function will automatically launch a PosDefexception that will be catched</p>
- In the catch, since it was impossible to factorize through the Cholesky, we perform a LDL^T factorization with the sparsity computed before.In addition, we specifically called the ldlt of the Ban module, implemented by us, in order to avoid to julia the task of discriminating which one should be called</p>

**Re-implementation of the operator **

For the purpose of resolving our system, we re-implemented the operator '\' used to automatically resolve the sytem f1 and f2.
The operator receives as input a BanFactor and, depending on the type of factorization it has been performed, it will called a function to either resolve the Cholesky or LDL^T system

In [ ]:

```
Base.:(\)(f_::BanFactor, b_::Vector) = (f_.flag == UInt(1)) ? _solve_cholesky(f_, b_) :
(f_.P == []) ? _solve_ldlt(f_, b_) : _solve_ldlt_perm(f_, b_)

function _solve_ldlt_perm( f::BanFactor, b::Vector )
    # A x == b
    # P * A * Pt = L*D*Lt
    # A = Pt* L*D *Lt *P

    # Pt* L*D *Lt *P * x == b
    # L* D* Lt * P * x == P * b

    # b1 = f.P * b
    b1 = b[f.P]

    y = f.L\b1
    # D* Lt * P * x = y
    z = f.D\y
    # Lt* P * x = z
    x1 = f.Lt\z

    #  x1 = P * x
    #  x = Pt * x1
    x = x1[invperm(f.P)]

    return x
end
function _solve_ldlt( f::BanFactor, b::Vector )
    # A x == b
    # L* D* Lt * x ==  b
    y = f.L\b
    # D* Lt * P * x = y
    z = f.D\y
    # Lt* P * x = z
    x = f.Lt\z

    return x
end
function _solve_cholesky( f::BanFactor, b::Vector )
    # A x == b
    # L* Lt * x ==  b

    b1 = b[f.P]

    y = f.L\b1
    x1 = f.Lt\y

    x = x1[invperm(f.P)]

    return x
end
```

Kyte LP problem

In [1]:

```
include("ipqp.jl")
#include("BAN.jl")
using .BAN
using LinearAlgebra
```

In [2]:

```
using BenchmarkTools
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 120
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

Out[2]:

```
10
```

In [3]:

```
Q = zeros(2,2)

#c = [-8-14η, -12];     # converges to (30, 50)
#c = [-8-4η, -12-10η];  # converges to (0, 70)
c = [-8, -12];  # converges to (30, 50)
#c = [-8, -12-10η];     # converges to (0, 70)

b = [120; 210; 270; 60];

A = [ 2  1;
      2  3;
      4  3;
     -1 -2];

A = [A I]

A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [9]:

```
println("--> BENCHMARK NOT-SPARSE...")
#b1 = @benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false)
@benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false, doPrint = f
alse,lex=false)
```

```
--> BENCHMARK NOT-SPARSE...
```

Out[9]:

```
BenchmarkTools.Trial:
  memory estimate:  26.43 MiB
  allocs estimate:  470127
  --------------
  minimum time:     40.932 ms (0.00% GC)
  median time:      46.243 ms (8.48% GC)
  mean time:        46.132 ms (5.38% GC)
  maximum time:     50.702 ms (7.47% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [8]:

```
@time ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false, doPrint = false,
lex=false);
```

```
  0.050291 seconds (419.91 k allocations: 25.396 MiB, 23.37% gc time)
```

In [12]:

```
println("--> BENCHMARK SPARSE...")
#b2 = @benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true)
@benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true,lex=false)
```

```
--> BENCHMARK SPARSE...
```

Out[12]:

```
BenchmarkTools.Trial:
  memory estimate:  27.27 MiB
  allocs estimate:  499210
  --------------
  minimum time:     37.917 ms (0.00% GC)
  median time:      44.419 ms (8.11% GC)
  mean time:        46.929 ms (5.05% GC)
  maximum time:     62.165 ms (7.04% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [11]:

```
@time ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true, doPrint = false,
lex=false) ;
```

```
  0.039742 seconds (431.39 k allocations: 25.904 MiB)
```

Kyte LP lex

In [1]:

```
include("ipqp.jl")
#include("BAN.jl")
using .BAN
using LinearAlgebra
```

In [10]:

```
Q = zeros(2,2)

#c = [-8-14η, -12];     # converges to (30, 50)
#c = [-8-4η, -12-10η];  # converges to (0, 70)
c = [-8-14η, -12-10η];  # converges to (30, 50)
#c = [-8, -12-10η];     # converges to (0, 70)

b = [120; 210; 270; 60];

A = [ 2  1;
      2  3;
      4  3;
     -1 -2];

A = [A I]

#A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [3]:

```
using BenchmarkTools
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 120
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

Out[3]:

```
10
```

In [24]:

```
println("--> BENCHMARK NOT-SPARSE...")
#b1 = @benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false)
@benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false,doPrint = fa
lse, lex=true)
```

```
--> BENCHMARK NOT-SPARSE...
```

Out[24]:

```
BenchmarkTools.Trial: 
  memory estimate:  61.09 MiB
  allocs estimate:  1003410
  --------------
  minimum time:     91.301 ms (4.30% GC)
  median time:      98.860 ms (4.16% GC)
  mean time:        97.707 ms (5.50% GC)
  maximum time:     103.110 ms (8.06% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [28]:

```
@time ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false,doPrint=false,le
x=true);
```

```
  0.106020 seconds (1.00 M allocations: 61.091 MiB, 4.73% gc time)
```

In [23]:

```
println("--> BENCHMARK SPARSE...")
#b2 = @benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true)
@benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true, lex=true)
```

```
--> BENCHMARK SPARSE...
```

Out[23]:

```
BenchmarkTools.Trial: 
  memory estimate:  60.19 MiB
  allocs estimate:  962730
  --------------
  minimum time:     83.425 ms (4.32% GC)
  median time:      87.589 ms (4.17% GC)
  mean time:        87.723 ms (5.34% GC)
  maximum time:     93.910 ms (3.77% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [19]:

```
@time ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true, doPrint = false,
lex=true);
```

```
  0.086365 seconds (962.73 k allocations: 60.186 MiB, 13.62% gc time)
```

Augumenting Kyte LP

In order to test how our program behave when we increase the sparsity on the matrix A, we implement a function,called 'augument_system'.
This function will add fictitious slack variables to the matrix A, namely some variables that we don't minimize,that doesn't appear on the function c, which b values assigned is equal to 1 (not a random value because in the standard solution with the LU factorization the system performed way worst, and had problem of convergence).

In [1]:

```
include("ipqp.jl")
#include("BAN.jl")
using .BAN
using LinearAlgebra
```

In [2]:

```
using Plots
```

In [3]:

```
using BenchmarkTools
using SparseArrays
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 120
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

Out[3]:

```
10
```

In [4]:

```
function augment_system(_A, _b, _c, _Q, num)
    nA, mA = size(_A)
    _A = [ _A zeros(nA, num)]
    _A = [_A ; zeros( num, mA ) I ]

    nQ, mQ = size(Q)
    _Q = [_Q zeros(nQ, num)]
    _Q = [_Q ; zeros(num, mQ + num)]

    for i in 1:num
        push!(_b, 1)
        push!(_c, 0)
    end
    return _A, _b, _c, _Q
end
```

Out[4]:

```
augment_system (generic function with 1 method)
```

Kyte LP standard

In [5]:

```
Q = zeros(2,2)

#c = [-8-14η, -12];      # converges to (30, 50)
#c = [-8-4η, -12-10η];  # converges to (0, 70)
c = [-8, -12];  # converges to (30, 50)
#c = [-8, -12-10η];      # converges to (0, 70)

b = [120; 210; 270; 60];

A = [ 2  1;
      2  3;
      4  3;
     -1 -2];

A = [A I]

A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [34]:

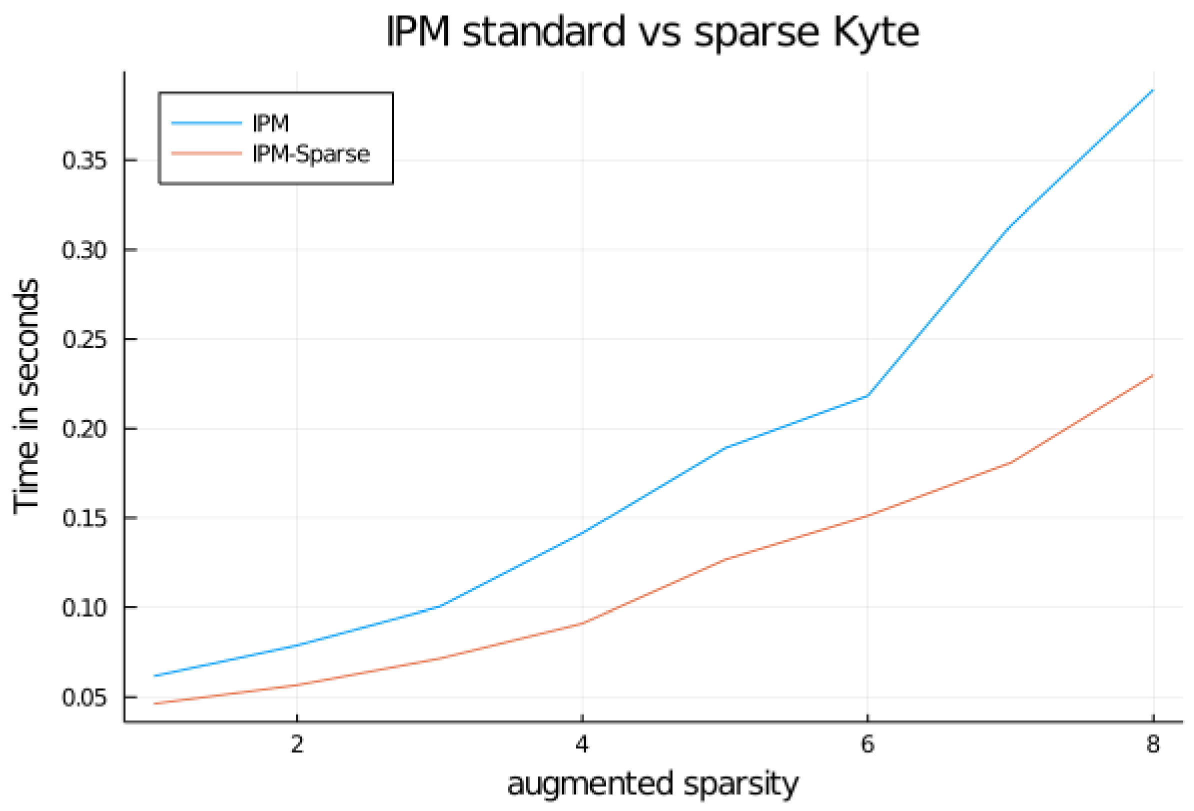
```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = true,doPrint=false
,lex=false);
```

```
  0.220482 seconds (3.08 M allocations: 219.815 MiB, 10.51% gc time)
```

In [55]:

```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = false,doPrint=fals
e,lex=false);
```

```
  0.407319 seconds (3.93 M allocations: 266.874 MiB, 10.20% gc time)
```



Kyte LP lex

In [1]:

```
include("ipqp.jl")
#include("BAN.jl")
using .BAN
using LinearAlgebra
```

In [56]:

```
Q = zeros(2,2)

#c = [-8-14η, -12];     # converges to (30, 50)
#c = [-8-4η, -12-10η];  # converges to (0, 70)
c = [-8-14η, -12-10η];  # converges to (30, 50)
#c = [-8, -12-10η];     # converges to (0, 70)

b = [120; 210; 270; 60];

A = [ 2  1;
      2  3;
      4  3;
     -1 -2];

A = [A I]

#A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [126]:

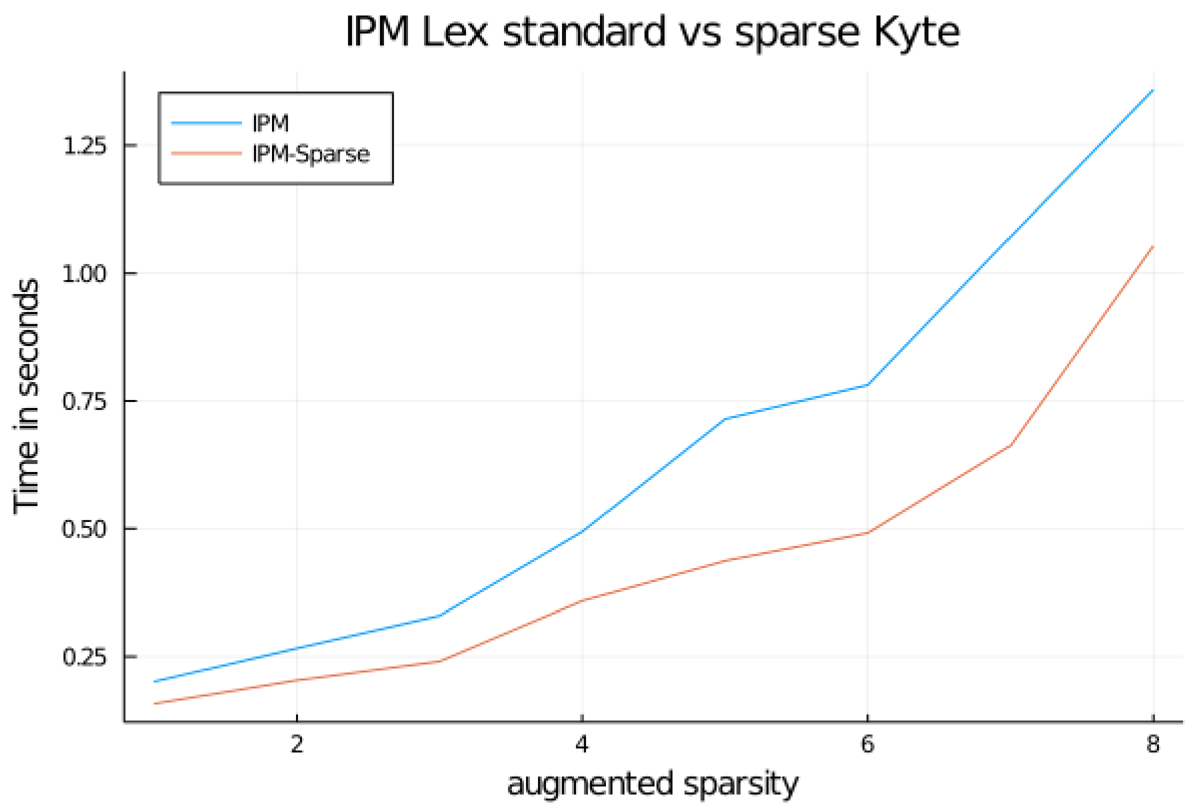
```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = true,doPrint=false
,lex=true);
```

```
  1.163729 seconds (9.88 M allocations: 662.597 MiB, 11.10% gc time)
```

In [124]:

```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = false,doPrint=fals
e,lex=true);
```

```
  1.260013 seconds (8.59 M allocations: 582.155 MiB, 10.86% gc time)
```



In the case of standard and lexicographic Kyte, we can notice than, indipendently from the augumentation to artificially increase the sparsity of the sytem, the system will always better in term of time performance. Not only, obviously, since we are working with sparse representation, we will have a better usage of the memory, less occupied and with fewer call to the garbage collector.
In other words, this approach looks to be without any disadvantages, but only positive sides.

3-objective quadratic problem

In [1]:

```
include("ipqp.jl")
```

Out[1]:

```
ipqp (generic function with 1 method)
```

In [2]:

```
using BenchmarkTools
using LinearAlgebra
using SparseArrays
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 120
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

Out[2]:

```
10
```

In [127]:

```
Q = [2 2 0;
     2 2 0;
     0 0 2]

c = [-1, -1, -1];
b = [1; 1; 1; 3];

A = [ -1   1   1;
      -1  -1   1;
       1  -1   1;
       1   1   1]

A = [A I]
#A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [13]:

```
println("--> BENCHMARK NOT-SPARSE...")
#b1 = @benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false)
@benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false,doPrint =fal
se,lex=false)
```

```
--> BENCHMARK NOT-SPARSE...
```

Out[13]:

```
BenchmarkTools.Trial:
  memory estimate:  33.71 MiB
  allocs estimate:  557574
  --------------
  minimum time:     81.004 ms (0.00% GC)
  median time:      87.576 ms (6.46% GC)
  mean time:        87.069 ms (4.64% GC)
  maximum time:     89.973 ms (6.65% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [16]:

```
@time ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false, doPrint = false,
lex=false);
```

```
  0.089261 seconds (557.56 k allocations: 33.711 MiB, 10.63% gc time)
```

In [12]:

```
println("--> BENCHMARK SPARSE...")
#b2 = @benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true)
@benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true,doPrint=fals
e,lex=false)
```

```
--> BENCHMARK SPARSE...
```

Out[12]:

```
BenchmarkTools.Trial:
  memory estimate:  28.49 MiB
  allocs estimate:  437398
  --------------
  minimum time:     57.425 ms (0.00% GC)
  median time:      62.170 ms (7.22% GC)
  mean time:        62.191 ms (4.45% GC)
  maximum time:     70.435 ms (6.71% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [19]:

```
@time ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true, doPrint=false,lex
=false);
```

```
  0.053524 seconds (437.40 k allocations: 28.495 MiB, 23.42% gc time)
```

3-objective quadratic problem lex

In [20]:

```
A = [ -1   1   1  1  0  0  0;
      -1  -1   1  0  1  0  0;
       1  -1   1  0  0  1  0;
       1   1   1  0  0  0  1]

b = [1; 1; 1; 3]

c = [-1, -1, -1,  0,  0, 0, 0]
q = [-5, -5,  0,  0,  0, 0 ,0]
p = [-5, -3,  2,  0,  0, 0, 0]

c = c + q*η #+ p*(η^2)

#     x1  x2  x3
Q = [ 2   2   0 ;
      2   2   0 ;
      0   0   2  ]
P = [4 0 0; 0 4 0; 0 0 0]

Q = Q + P*η

Q = [Q zeros(3,4); zeros(4, 7)]
```

Out[20]:

```
7×7 Matrix{Ban}:
 α^0(2.0 + 4.0η^1 + 0.0η^2)  …  α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(2.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)  …  α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
```

In [7]:

```
println("--> BENCHMARK NOT-SPARSE...")
#b1 = @benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false)
@benchmark ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false,doPrint=fals
e,lex=true)
```

```
--> BENCHMARK NOT-SPARSE...
```

Out[7]:

```
BenchmarkTools.Trial:
  memory estimate:  685.30 MiB
  allocs estimate:  11799394
  --------------
  minimum time:     1.354 s (5.97% GC)
  median time:      1.523 s (5.71% GC)
  mean time:        1.543 s (5.55% GC)
  maximum time:     1.830 s (5.08% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [22]:

```
@time ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = false,doPrint=false,lex
=true);
```

```
  1.393061 seconds (11.77 M allocations: 684.804 MiB, 10.16% gc time)
```

In [8]:

```
println("--> BENCHMARK SPARSE...")
#b2 = @benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true)
@benchmark ipqp( copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true,doPrint=fals
e,lex=true)
```

```
--> BENCHMARK SPARSE...
```

Out[8]:

```
BenchmarkTools.Trial:
  memory estimate:  74.84 MiB
  allocs estimate:  1180968
  --------------
  minimum time:     153.383 ms (2.97% GC)
  median time:      162.842 ms (6.03% GC)
  mean time:        163.164 ms (5.14% GC)
  maximum time:     172.114 ms (2.84% GC)
  --------------
  samples:          10
  evals/sample:     1
```

In [25]:

```
@time ipqp(copy(A), copy(b), copy(c), copy(Q), 1e-8; sparsity = true, doPrint=false,lex
=true);
```

```
  0.134362 seconds (1.18 M allocations: 74.843 MiB, 8.18% gc time)
```

Augument 3-objective quadratic problem

In [1]:

```
include("ipqp.jl")
```

Out[1]:

```
ipqp (generic function with 1 method)
```

In [2]:

```
using Plots
```

In [3]:

```
using BenchmarkTools
using LinearAlgebra
using SparseArrays
BenchmarkTools.DEFAULT_PARAMETERS.seconds = 120
BenchmarkTools.DEFAULT_PARAMETERS.samples = 10
```

Out[3]:

```
10
```

In [4]:

```
function augment_system(_A, _b, _c, _Q, num)
    nA, mA = size(_A)
    _A = [ _A zeros(nA, num)]
    _A = [_A ; zeros( num, mA ) I ]

    nQ, mQ = size(Q)
    _Q = [_Q zeros(nQ, num)]
    _Q = [_Q ; zeros(num, mQ + num)]

    for i in 1:num
        push!(_b, 1)
        push!(_c, 0)
    end
    return _A, _b, _c, _Q
end
```

Out[4]:

```
augment_system (generic function with 1 method)
```

3-objective quadratic problem

In [128]:

```
Q = [2 2 0;
     2 2 0;
     0 0 2]

c = [-1, -1, -1];
b = [1; 1; 1; 3];

A = [ -1   1   1;
      -1  -1   1;
       1  -1   1;
       1   1   1]

A = [A I]
#A = convert(Matrix{Ban}, A);

c = vcat(c, zeros(size(A,1)))
Q = [Q zeros(size(Q,1), size(A,1)); zeros(size(A,1), size(Q,1)+size(A,1))]

tol=1e-8;
```

In [166]:

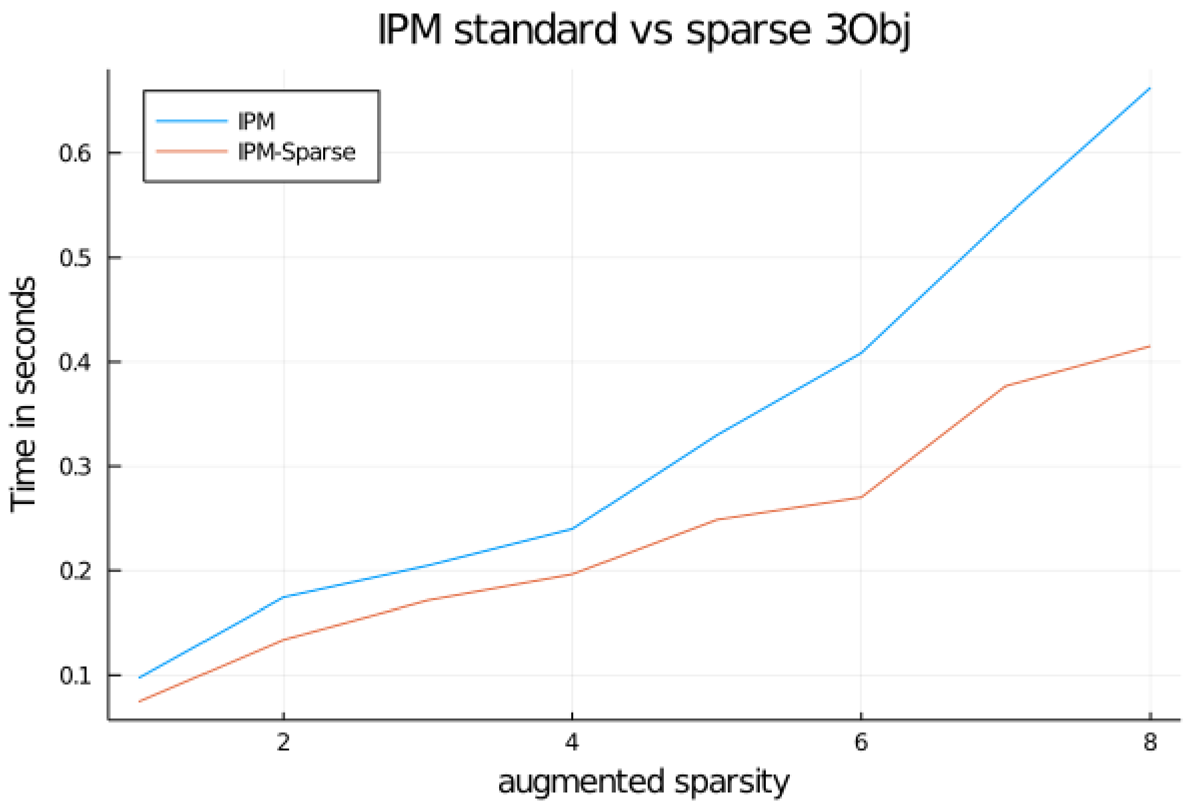
```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = true,doPrint=false
,lex=false);
```

```
  0.415166 seconds (3.19 M allocations: 215.040 MiB, 11.47% gc time)
```

In [183]:

```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = false,doPrint=fals
e,lex=false);
```

```
  0.671096 seconds (4.46 M allocations: 302.379 MiB, 12.25% gc time)
```



3-objective quadratic problem lex

In [185]:

```
A = [ -1   1   1  1  0  0  0;
      -1  -1   1  0  1  0  0;
       1  -1   1  0  0  1  0;
       1   1   1  0  0  0  1]

b = [1; 1; 1; 3]

c = [-1, -1, -1,  0,  0, 0, 0]
q = [-5, -5,  0,  0,  0, 0 ,0]
p = [-5, -3,  2,  0,  0, 0, 0]

c = c + q*η #+ p*(η^2)

#     x1  x2  x3
Q = [ 2   2   0 ;
      2   2   0 ;
      0   0   2  ]
P = [4 0 0; 0 4 0; 0 0 0]

Q = Q + P*η

Q = [Q zeros(3,4); zeros(4, 7)]
```

Out[185]:

```
7×7 Matrix{Ban}:
 α^0(2.0 + 4.0η^1 + 0.0η^2)  …  α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(2.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)  …  α^0(0.0 + 0.0η^1 + 0.0η^2)
 α^0(0.0 + 0.0η^1 + 0.0η^2)     α^0(0.0 + 0.0η^1 + 0.0η^2)
```

In [221]:

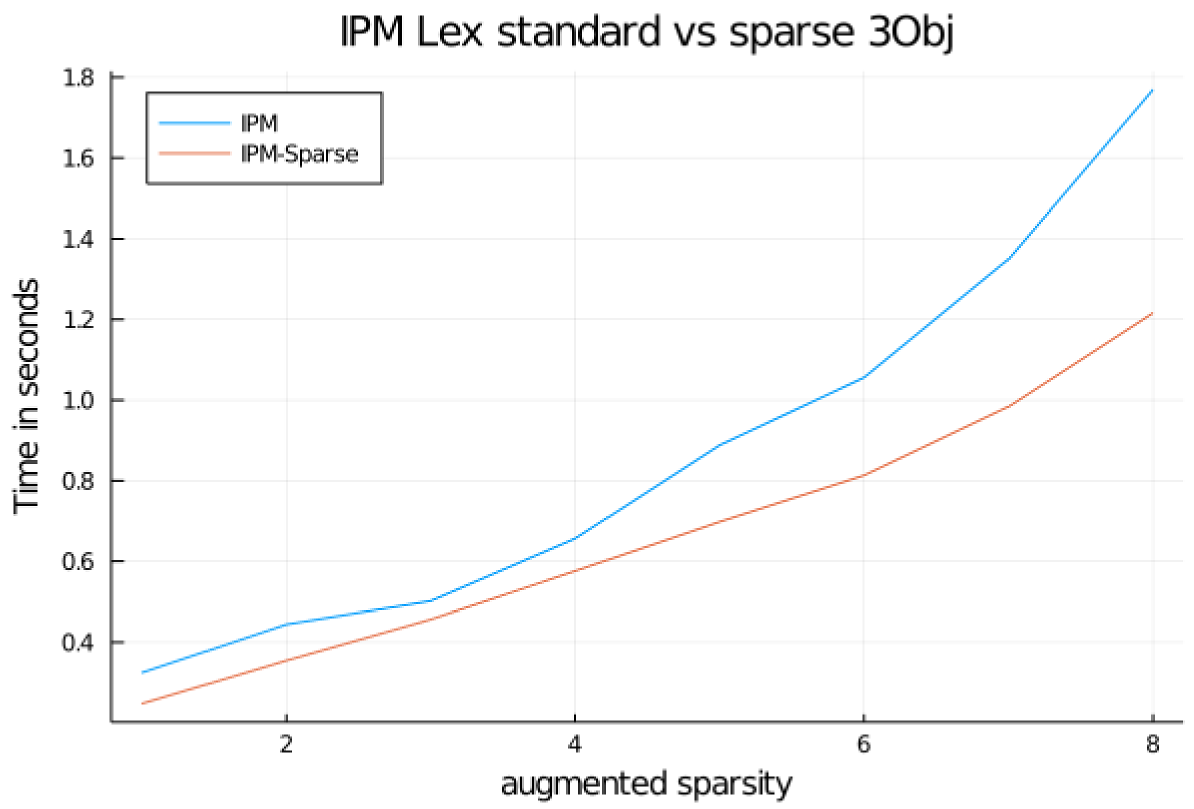
```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = true,doPrint=false
,lex=true);
```

```
  1.277029 seconds (8.45 M allocations: 567.005 MiB, 12.08% gc time)
```

In [217]:

```
A1, b1, c1, Q1 = copy(A), copy(b), copy(c), copy(Q)
A1, b1, c1, Q1 = augment_system(copy(A), copy(b), copy(c), copy(Q), 8)
@time ipqp( copy(A1), copy(b1), copy(c1), copy(Q1), 1e-8; sparsity = false,doPrint=fals
e,lex=true);
```

```
  1.687998 seconds (9.97 M allocations: 672.419 MiB, 12.23% gc time)
```



As we notice before with the Kyte problem, the same observations can be made even for the 3-objective quadratic problem:indipendently from the augumentation to artificially increase the sparsity of the sytem, the system will always better in term of time performance and memory usage.

LP_AFIRO

27x51 102 - Nonzeros

In [1]:

```
include("ipqp.jl")
using MatrixDepot
```

```
include group.jl for user defined matrix generators
verify download of index files...
reading database
adding metadata...
adding svd data...
writing database
used remote sites are sparse.tamu.edu with MAT index and math.nist.gov wit
h HTML index
```

In [2]:

```
sparsity = 1-(102/(27*51))
```

Out[2]:

```
0.9259259259259259
```

In [3]:

```
println("Loading problem...")
tol = 1e-8
name = "LPnetlib/lp_afiro"
try global P = mdopen(name)
catch
    global P = mdopen(name)
end
```

```
Loading problem...
```

Out[3]:

```
(RG LPnetlib/lp_afiro(#597)  27x51(102)  [A, b, c, hi, lo, z0] 'linear pro
gramming problem' [Netlib LP problem afiro: minimize c'*x, where Ax=b, lo<
=x<=hi])()
```

In [6]:

```
println("--> BENCHMARK NOT-SPARSE...")
@time ipqp(copy(P.A), vec(copy(P.b)), vec(copy(P.c)), zeros( size(copy(P.c))[1], size(c
opy(P.c))[1]  ), tol; maxit=20, sparsity=false, doPrint=true,lex=false);
```

```
--> BENCHMARK NOT-SPARSE...
  2.368526 seconds (21.74 M allocations: 1.452 GiB, 17.49% gc time)
  2.399484 seconds (21.74 M allocations: 1.452 GiB, 17.69% gc time)
  2.487797 seconds (21.83 M allocations: 1.455 GiB, 18.03% gc time)
  2.555318 seconds (21.86 M allocations: 1.456 GiB, 18.25% gc time)
  2.523785 seconds (21.95 M allocations: 1.458 GiB, 17.91% gc time)
  2.379785 seconds (22.17 M allocations: 1.464 GiB, 16.86% gc time)
  2.413281 seconds (22.09 M allocations: 1.462 GiB, 14.55% gc time)
  2.417275 seconds (21.74 M allocations: 1.453 GiB, 15.28% gc time)
  2.395509 seconds (21.68 M allocations: 1.451 GiB, 16.38% gc time)
 22.968013 seconds (205.90 M allocations: 13.703 GiB, 16.91% gc time)
```

In [7]:

```
println("--> BENCHMARK SPARSE...")
@time ipqp(copy(P.A), vec(copy(P.b)), vec(copy(P.c)), zeros( size(copy(P.c))[1], size(c
opy(P.c))[1]  ), tol; maxit=20, sparsity=true, doPrint=true,lex=false);
```

```
--> BENCHMARK SPARSE...
  0.042398 seconds (113.20 k allocations: 8.292 MiB, 20.98% gc time)
  1.226701 seconds (10.92 M allocations: 747.956 MiB, 16.29% gc time)
  1.279639 seconds (10.92 M allocations: 747.956 MiB, 16.27% gc time)
  1.188729 seconds (10.92 M allocations: 747.956 MiB, 16.89% gc time)
  1.294717 seconds (10.92 M allocations: 747.956 MiB, 17.27% gc time)
  1.241660 seconds (10.92 M allocations: 747.956 MiB, 16.66% gc time)
  1.279059 seconds (10.92 M allocations: 747.956 MiB, 17.30% gc time)
  1.245990 seconds (10.92 M allocations: 747.956 MiB, 18.15% gc time)
  1.256920 seconds (10.92 M allocations: 747.956 MiB, 17.25% gc time)
 12.193302 seconds (112.49 M allocations: 7.652 GiB, 17.73% gc time)
```

lpi_forest6

66x131 246 - Nonzeros

In [8]:

```
sparsity = 1- (246/(66*131))
```

Out[8]:

```
0.9715475364330326
```

In [9]:

```
println("Loading problem...")
tol = 1e-8
name = "LPnetlib/lpi_forest6"
try global P = mdopen(name)
catch
    global P = mdopen(name)
end
```

```
Loading problem...
```

Out[9]:

```
(RG LPnetlib/lpi_forest6(#714)  66x131(246) 1993 [A, b, c, hi, lo, z0] 'li
near programming problem' [Netlib LP problem forest6: minimize c'*x, where
Ax=b, lo<=x<=hi])()
```

In [10]:

```
println("--> BENCHMARK NOT-SPARSE...")
@time ipqp(copy(P.A), vec(copy(P.b)), vec(copy(P.c)), zeros( size(copy(P.c))[1], size(c
opy(P.c))[1]  ), tol; maxit=20, sparsity=false,doPrint=true,lex=false);
```

```
--> BENCHMARK NOT-SPARSE...
 47.814366 seconds (355.54 M allocations: 23.786 GiB, 25.72% gc time)
 50.081472 seconds (355.55 M allocations: 23.787 GiB, 26.38% gc time)
 49.674243 seconds (355.63 M allocations: 23.790 GiB, 26.34% gc time)
 48.746592 seconds (355.76 M allocations: 23.793 GiB, 26.26% gc time)
 50.941398 seconds (356.14 M allocations: 23.805 GiB, 25.99% gc time)
 50.937043 seconds (355.96 M allocations: 23.799 GiB, 25.71% gc time)
 52.431859 seconds (356.90 M allocations: 23.828 GiB, 25.78% gc time)
 50.064362 seconds (355.07 M allocations: 23.778 GiB, 26.35% gc time)
 49.727210 seconds (354.39 M allocations: 23.753 GiB, 26.37% gc time)
 49.666227 seconds (354.17 M allocations: 23.743 GiB, 26.38% gc time)
507.021087 seconds (3.62 G allocations: 241.979 GiB, 26.03% gc time)
```

In [12]:

```
println("--> BENCHMARK SPARSE...")
@time ipqp(copy(P.A), vec(copy(P.b)), vec(copy(P.c)), zeros( size(copy(P.c))[1], size(c
opy(P.c))[1] ), tol; maxit=20, sparsity=true,doPrint=true,lex=false);
```

```
--> BENCHMARK SPARSE...
  0.073994 seconds (1.24 M allocations: 99.678 MiB, 22.86% gc time)
 22.327901 seconds (182.25 M allocations: 12.220 GiB, 23.95% gc time)
 23.786499 seconds (182.25 M allocations: 12.220 GiB, 25.03% gc time)
 25.516298 seconds (182.25 M allocations: 12.220 GiB, 21.34% gc time)
 29.657546 seconds (182.25 M allocations: 12.220 GiB, 22.19% gc time)
 32.013811 seconds (182.25 M allocations: 12.220 GiB, 20.51% gc time)
 25.759107 seconds (182.25 M allocations: 12.220 GiB, 23.38% gc time)
 25.104931 seconds (182.25 M allocations: 12.220 GiB, 21.26% gc time)
 24.471081 seconds (182.25 M allocations: 12.220 GiB, 23.90% gc time)
 15.870544 seconds (116.47 M allocations: 8.476 GiB, 22.53% gc time)
253.201278 seconds (1.89 G allocations: 130.024 GiB, 23.09% gc time)
```