Efficient GMRES method for solving particular KKT systems for quadratic flow optimization

Daniele Di Sarli
daniele.disarli@gmail.com

Shadi Shajari
shajari.shadi@gmail.com

January 14, 2018

A.Y. 2017/2018

MNO Course (371AA) Project number: 13 w/o ML

Abstract

We describe our MATLAB implementation of GMRES, an iterative method for the efficient solution of systems of linear equations. Our method is focused on the efficient resolution of a special kind of matrices coming from quadratic min-flow cost problems. Within the algorithm, we make use of a custom implementation of Arnoldi iteration and QR factorization. We conclude with some numerical experiments for the analysis of the performance.

1 Introduction

We show an algorithm for the resolution of large, sparse linear systems with the following structure:

$$\begin{bmatrix} D & E^T \\ E & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix} \tag{1}$$

where D is a diagonal matrix with positive entries and E is the edge-node adjacency matrix of a graph. They are combined to produce a matrix A of size $m \times m$.

These kind of systems arise as the KKT system of certain quadratic flow optimization problems on graphs. In particular, E is the edge-node adjacency matrix of a graph: its structure can be exploited as we have only two non-zero elements for each column, and their values are exactly 1 and -1.

To solve the problem we will use the GMRES algorithm from Saad and Schultz [1], exploiting a specialized version of QR factorization for Hessenberg matrices as an inner step. This will bring down the cost of the inner step from $O(k^3)$ to $O(k^2)$.

We will expose the problem by following a top-down approach. In the last section, we show some analysis of the performance and optimizations.

2 Entry point: solvels

The entry point of our algorithm, called *solvels* for “solve linear system", will have the following data as input:

- $\boldsymbol{d}$: a vector containing the diagonal entries of D, of size $(m-z)\times 1$
- E: the sparse matrix of size $z \times (m-z)$
- $\boldsymbol{b}$: the column vector of the linear system, of size $m-z$
- $\boldsymbol{c}$: the column vector of the linear system, of size z
- max_iterations: stopping criterion relative to the number of iterations
- tol: stopping criterion relative to the tolerance the user wants to achieve.

The output is just the solution $\boldsymbol{x}$.

Here we observe that instead of running GMRES on the original problem (1), we can just find the solution of a smaller linear system.

First, notice that our linear system is equivalent to:

$$\begin{cases} D\boldsymbol{x} + E^T\boldsymbol{y} = \boldsymbol{b} \\ E\boldsymbol{x} = \boldsymbol{c} \end{cases}$$

From the first equation, we get

$$\boldsymbol{x} = D^{-1}(\boldsymbol{b} - E^T\boldsymbol{y}) = D^{-1}\boldsymbol{b} - D^{-1}E^T\boldsymbol{y} \tag{2}$$

And by substituting $\boldsymbol{x}$ in the second equation, we obtain

$$\begin{aligned} E(D^{-1}\boldsymbol{b} - D^{-1}E^T\boldsymbol{y}) &= \boldsymbol{c} \\ (ED^{-1}E^T)\boldsymbol{y} &= ED^{-1}\boldsymbol{b} - \boldsymbol{c} \end{aligned} \tag{3}$$

so we can use GMRES to find the value of $\boldsymbol{y}$, and then simply substitute it in Equation 2 to find $\boldsymbol{x}$.

Note that the matrix of this new linear system is generally smaller than A ($z \times z$ instead of $m \times m$), and is symmetric.

The pseudocode for the function is shown in Algorithm 1. We also included some optimization based on element-wise operations in order to, for example, avoid computing the inverse of $D = \operatorname{diag}(d)$ through generic (possibly inefficient) algorithms.

Algorithm 1 Entry point for the software

```
1: function SOLVELS($\boldsymbol{d}, E, \boldsymbol{b}, \boldsymbol{c}$, max_iterations, tol)
2:                                   ▷ $\odot$ and $\oslash$ are resp. element-wise product and division
3:     ▷ Instead of using the library calls to compute $D^{-1}$, we exploit the fact that it
   is diagonal and we produce a sparse matrix.
4:     Dinv ← sparse_diag($1 \oslash \boldsymbol{d}$)    ▷ $\text{Dinv} = D^{-1} = \begin{bmatrix} \frac{1}{d_1} & 0 & \dots & 0 \\ 0 & \frac{1}{d_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{d_{m-z}} \end{bmatrix}$
5:     EDinv ← $E *$ Dinv
6:     $\tilde{A}$ ← EDinv $* E^T$                        ▷ $\tilde{A} = ED^{-1}E^T$
7:     $\tilde{\boldsymbol{b}}$ ← EDinv $* \boldsymbol{b} - \boldsymbol{c}$                    ▷ $\tilde{\boldsymbol{b}} = ED^{-1}\boldsymbol{b} - \boldsymbol{c}$
8:     $f \leftarrow \lambda \boldsymbol{v}.\tilde{A}\boldsymbol{v}$
9:                                   ▷ Solve the linear system to find y
10:    $\boldsymbol{y}$ ← GMRES($f, \tilde{\boldsymbol{b}}$, max_iterations, tol)
11:    ▷ now we have $\boldsymbol{x} = D^{-1}(\boldsymbol{b} - E^T\boldsymbol{y})$, but we can avoid going through the square
   matrix $D^{-1}$
12:    $\boldsymbol{x} \leftarrow (1 \oslash \boldsymbol{d}) \odot (\boldsymbol{b} - E^T\boldsymbol{y})$
13:    return $\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$
```

3 GMRES

We use GMRES [1] to find the solution of the linear system of Equation 3.

This is an iterative method that solves

$$\min_x \|A\boldsymbol{x} - \boldsymbol{b}\|, \quad \boldsymbol{x} \in \mathcal{K}_n(A, \boldsymbol{b}) \tag{4}$$

where $\mathcal{K}_n(A, b) = \text{span}\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, \dots, A^{n-1}\boldsymbol{b}\}$ is the n-th order Krylov subspace generated by $A \in \mathbb{R}^{z \times z}$ and $\boldsymbol{b} \in \mathbb{R}^z$. [1]

Assume that, at step n, we have Q_n and $\underline{H}_n$ such that $AQ_n = Q_{n+1}\underline{H}_n$. They can be generated e.g. from an Arnoldi iteration (described later). In particular, Q_n is a $z \times n$ matrix whose columns are the orthonormal vectors that span $\mathcal{K}_n(A, b)$, and H is a $(n+1) \times n$ upper Hessenberg matrix augmented with one additional row.

Then, because $Q_n y \in \mathcal{K}_n(A, b)$, our Equation 4 becomes equivalent to

$$\min_y \|A(Q_n\boldsymbol{y}) - \boldsymbol{b}\|$$

We can derive

$$\|AQ_n\boldsymbol{y} - \boldsymbol{b}\| = \|Q_{n+1}\underline{H}_n\boldsymbol{y} - \boldsymbol{b}\| = \|\underline{H}_n\boldsymbol{y} - \|\boldsymbol{b}\|\, \boldsymbol{e}_1\|$$

[1]From now on, for clarity of notation, we overload the symbol A to indicate a new, generic matrix; i.e., this is not the same A from Section 2.

So to find y we can just minimize

$$\min_{y} \left\| \underline{H}_n \boldsymbol{y} - \|\boldsymbol{b}\| \, \boldsymbol{e}_1 \right\|$$

where $\underline{H}_n \in \mathbb{R}^{(n+1)\times n}$: this is a much smaller problem than (4).

Finding y. To find $\boldsymbol{y}$, it is convenient to solve the linear system with the help of a QR factorization of $\underline{H}_n$, which can be optimized knowing the special structure of $\underline{H}_n$.

Assume we computed the QR factorization of $\underline{H}_n$, which produced $\widetilde{Q} \in \mathbb{R}^{(n+1)\times(n+1)}$ and $\widetilde{R} \in \mathbb{R}^{(n+1)\times n}$. Consider this decomposition:

$$\widetilde{Q} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$$

$$\widetilde{R} = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

where $Q_1 \in \mathbb{R}^{(n+1)\times n}$ and $R_1 \in \mathbb{R}^{n\times n}$. We want to find a $\boldsymbol{y}$ that minimizes $\left\| \underline{H}_n \boldsymbol{y} - \|\boldsymbol{b}\| \, \boldsymbol{e}_1 \right\|_2$, so we take

$$\boldsymbol{y} = R_1^{-1} Q_1^T \boldsymbol{b} \tag{5}$$

At the end, we can finally compute $\boldsymbol{x}^{(n)} = Q_n \boldsymbol{y}$.

Residual. At the end of each step, we would like to compute the current error with respect to the optimal solution $\bar{\boldsymbol{x}}$. We can do that by computing the residual

$$r_n = \frac{\left\| A\boldsymbol{x}^{(n)} - \boldsymbol{b} \right\|}{\|\boldsymbol{b}\|}$$

however, to avoid the expensive multiplication with A, we can compute the residual as: [2][3]

$$r_n = \frac{h_{n+1,n} \left| \boldsymbol{e}_n \boldsymbol{y} \right|}{\|\boldsymbol{b}\|} \tag{6}$$

where $h_{i,j}$ refers to the element of $\underline{H}_n$ indicated by the indices. In Figure A.1 it is shown the difference between the two residuals.

We perform an indefinite number of steps of the described algorithm, until the stopping criterions (a bound on the residual or a limit on the number of iterations) are satisfied.

We show the pseudocode for the GMRES function in Algorithm 2.

Algorithm 2 GMRES

```
function GMRES($A, \boldsymbol{b}$, max_iterations, tol)
    residual $\leftarrow -\infty$
    $n \leftarrow 1$
    while residual $>$ tol and $n <$ max_iterations do
        $Q_n, \underline{H}_n$, stop $\leftarrow$ ARNOLDI($A, \boldsymbol{b}, n$)
                                        ▷ Find $\boldsymbol{y}$ that minimizes $\|\underline{H}_n\boldsymbol{y} - \|\boldsymbol{b}\|\,\boldsymbol{e}_1\|_2$
        $\widetilde{Q}, \widetilde{R} \leftarrow$ HESSENBERG_QR($\underline{H}_n$)
        $\boldsymbol{y} \leftarrow \widetilde{R}(1{:}n, :)^{-1}\widetilde{Q}(:, 1{:}n)^T \,\|\boldsymbol{b}\|\,\boldsymbol{e}_1$
                                        ▷ Compute the approximated value of $\boldsymbol{x}$
        $\boldsymbol{x} \leftarrow Q_n(:, 1{:}n)\boldsymbol{y}$
        residual $\leftarrow \frac{1}{\|\boldsymbol{b}\|}\underline{H}_n(n{+}1, n)\,|\boldsymbol{e}_n\boldsymbol{y}|$
                                        ▷ Stop in case of Arnoldi breakdown
        if stop then break
        $n \leftarrow n + 1$
    return $\boldsymbol{x}$
```

4 Arnoldi iteration

At each step n of GMRES, we make use of the Arnoldi iteration in order to produce a basis for $\mathcal{K}_n(A, \boldsymbol{b})$. Note that the outputs of Arnoldi, which are Q_n and $\underline{H}_n$, can be computed immediately if Q_{n-1} and $\underline{H}_{n-1}$ are available. For this reason, at each iteration of GMRES we keep the outputs of Arnoldi and we pass them back to it at the next iteration.

The only (trivial) changes in GMRES are the initialization of the two matrices and their addition to the parameter list, as shown in Algorithm 3.

Algorithm 3 GMRES Patch for Arnoldi

```
...
    $Q_n \leftarrow [\,]$
    $\underline{H}_n \leftarrow [\,]$
    ...
    while ... do
        $Q_n, \underline{H}_n$, stop $\leftarrow$ ARNOLDI($A, \boldsymbol{b}, Q_n, \underline{H}_n, n$)
    ...
```

In this way, at each step n we can avoid doing a nested loop in the Arnoldi algorithm, and only concentrate on the generation of the last column of Q_n. Our inputs are:

- A: the sparse matrix of the linear system, of size $z \times z$
- $\boldsymbol{b}$: the column vector of the linear system, of size z

- Q_{n-1}: the Q matrix of size $z \times (n-1)$ from the previous iteration, or an empty matrix if $n = 1$.
- $\underline{H}_{n-1}$: the $\underline{H}$ matrix of size $n \times (n-1)$ from the previous iteration, or an empty matrix if $n = 1$.
- n: the number of iterations to perform

The theoretical properties stay the same: to build $\boldsymbol{q}_n$ we start by generating a vector $\boldsymbol{w}$ s.t. $\boldsymbol{w} \in \mathcal{K}_n(A, \boldsymbol{b}), \boldsymbol{w} \notin \mathcal{K}_{n-1}(A, \boldsymbol{b})$:

$$\boldsymbol{w} = A\boldsymbol{q}_{n-1} = \beta_1 \boldsymbol{q}_1 + \beta_2 \boldsymbol{q}_2 + \cdots + \beta_{n-1}\boldsymbol{q}_{n-1} + \beta_n \boldsymbol{q}_n$$

We notice that, $\forall i$, we have (since the basis is orthonormal):

$$\boldsymbol{q}_i^T \boldsymbol{w} = \beta_1 \underbrace{(\boldsymbol{q}_i^T \boldsymbol{q}_1)}_{=0} + \beta_2 \underbrace{(\boldsymbol{q}_i^T \boldsymbol{q}_2)}_{=0} + \cdots + \beta_i \underbrace{(\boldsymbol{q}_i^T \boldsymbol{q}_i)}_{=1} + \cdots + \beta_n \underbrace{(\boldsymbol{q}_i^T \boldsymbol{q}_n)}_{=0} = \beta_i$$

Hence we can compute

$$\beta_1 = \boldsymbol{q}_1^T \boldsymbol{w}, \quad \beta_2 = \boldsymbol{q}_2^T \boldsymbol{w}, \quad \ldots, \quad \beta_{n-1} = \boldsymbol{q}_{n-1}^T \boldsymbol{w}$$

and, subtracting, we get

$$\boldsymbol{g} := \beta_n \boldsymbol{q}_n = \boldsymbol{w} - \beta_1 \boldsymbol{q}_1 - \beta_2 \boldsymbol{q}_2 - \cdots - \beta_{n-1}\boldsymbol{q}_{n-1}$$

Finally, we want $\boldsymbol{q}_n$ to have norm 1, so we set:

$$\boldsymbol{q}_n = \frac{1}{\|\boldsymbol{g}\|}\boldsymbol{g}, \quad \beta_n = \|\boldsymbol{g}\|$$

At the end, the algorithm will produce in output the two matrices Q_n and $\underline{H}_n$, where Q_n is a $z \times n$ matrix whose columns are the orthonormal vectors that span $K_n(A, b)$, and $\underline{H}_n$ is a $(n+1) \times n$ upper Hessenberg matrix augmented with one additional row, which contains all the β_i of each iteration.

Lanczos. In our problem, the matrix A is symmetric (see Section 2). This means that also H_n, which is $\underline{H}_n$ without the last row, must be symmetric.

Proof. Let $h_{i,j}$ be the element of H_n at the specified indices, which by definition contains the β_i produced at iteration j. Then we have:

$$h_{j,i} = \boldsymbol{q}_j^T \boldsymbol{w} = \boldsymbol{q}_j^T A \boldsymbol{q}_i = (\boldsymbol{q}_j^T A \boldsymbol{q}_i)^T = \boldsymbol{q}_i^T A \boldsymbol{q}_j = h_{i,j}$$

□

If H_n is symmetric, it immediately follows that it must also be tridiagonal. Thus, we can improve the performance by reducing to the Lanczos algorithm [4], that shortens the orthogonalization loop of Arnoldi by iterating only over $[n-1, n]$ instead of $[1 \ldots n]$.

We show the pseudocode for the Arnoldi iteration (actually Lanczos) in Algorithm 4.

Algorithm 4 Arnoldi iteration

```
1: function ARNOLDI($A, \boldsymbol{b}, Q_{old}, \underline{H}_{old}, n, \text{tol}$)   ▷ $A$ is an $m \times m$ matrix, $\boldsymbol{b}$ is a vector of size $m$
2:     $H \leftarrow$ if empty($H_{old}$) then $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ else $\begin{bmatrix} H_{old} & 0 \\ 0 & 0 \end{bmatrix}$   ▷ an $(n+1) \times n$ matrix
3:     $Q \leftarrow \begin{bmatrix} Q_{old} & 0 \end{bmatrix}$   ▷ an $m \times n$ matrix
4:                                  ▷ Gram–Schmidt orthogonalization for the new column of $Q$
5:     $\boldsymbol{w} \leftarrow AQ(:, n)$
6:     for $i = \max(1, n-1)$ to $n$ do
7:         $H(i, n) \leftarrow Q(:, i)^T \boldsymbol{w}$
8:         $\boldsymbol{w} \leftarrow \boldsymbol{w} - H(i, n)Q(:, i)$
9:                                  ▷ Store the results for this iteration
10:    $H(n+1, n) = \|\boldsymbol{w}\|$
11:    breakdown $\leftarrow \|H(n+1, n)\| <$ tol   ▷ Check for breakdown
12:    $Q(:, n+1) = \boldsymbol{w} / \|\boldsymbol{w}\|$
13:    return $Q, H,$ breakdown
```

Cost. We have a matrix-vector multiplication on line 5 with cost $O(v)$ [2]. Within the loop we perform two multiplications, both with cost $O(n)$. We repeat the loop a constant number of times, so the cost of the whole function is $O(v + n)$.

5 QR Factorization for Hessenberg matrices

Inside GMRES we need to produce a QR factorization of $\underline{H}_n$ in order to solve the linear system of Equation 5.

We can exploit the fact that we know $\underline{H}_n$ to be upper Hessenberg with an additional row in order to produce a more efficient computation.

For simplicity of notation, let's overload again the symbol A and write $\underline{A} := \underline{H}_n$. The algorithm starts with the $(n+1) \times n$ input matrix $\underline{A}$ which is upper Hessenberg with an additional row. We want to compute the factorization $\underline{A} = QR$.

$$\underline{A} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}$$

The goal is to transform $\underline{A}$ into an upper triangular matrix with the use of Givens rotations, that will produce an orthogonal transformation.

[2] The multiplication could be more efficient than $O(m^2 n)$ thanks to the special structure of the matrix.

The transformed $\underline{A}$ will end up being our R, and the Givens rotations will produce our Q.

We iterate over the columns of $\underline{A}$, considering both the element on the diagonal and the one just below it. At each iteration i, we want to compute a rotation that will produce a zero below the diagonal.

To do that, we use a rotation matrix like the following, with an appropriate value for θ:

$$\hat{G} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

and we embed it into a $(n+1) \times (n+1)$ identity matrix G_i, at columns i, $i+1$ and rows i, $i+1$. Then we compute $\underline{A}_i = G_i \underline{A}_{i-1}$.

For example, after the first step we will produce (where the elements in bold are the only ones that changed):

$$\underline{A}_1 = \begin{bmatrix} \boldsymbol{*} & \boldsymbol{*} & \boldsymbol{*} & \boldsymbol{*} & \boldsymbol{*} \\ \mathbf{0} & \boldsymbol{*} & \boldsymbol{*} & \boldsymbol{*} & \boldsymbol{*} \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}$$

$$G_1 = \begin{bmatrix} \hat{G} & 0 \\ 0 & I \end{bmatrix}$$

such that $G_1 \underline{A} = \underline{A}_1$

We continue for the next columns of this transformed version of A, so that at the end of this iterative process we have:

$$G_n ... G_2 G_1 A = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Thus, we produced $R = (G_n ... G_2 G_1) A$ and so, by rearranging the terms, Q is just

$$Q = (G_n ... G_2 G_1)^{-1} = (G_n ... G_2 G_1)^T$$

Building the rotation matrix. For the rotation matrix $\hat{G}$, we would have to compute the value of θ to produce a zero below the diagonal of the matrix $\underline{A}$:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

Instead of explicitly computing θ we can observe that, because we're applying a rotation:

$$\left\| \begin{bmatrix} a \\ b \end{bmatrix} \right\| = \left\| \begin{bmatrix} r \\ 0 \end{bmatrix} \right\| = \|r\| \quad \implies \quad r = \sqrt{a^2 + b^2}$$

and then we can directly compute

$$c = a/r$$
$$s = -b/r$$

to get

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

This avoids going through the computation of the arctangent and the other trigonometric functions.

Multiplications of Q and R. We pointed out that, at each step, the multiplication $\underline{A}_i = G_i \underline{A}_{i-1}$ only affected the two rows i and $i+1$ of $\underline{A}_{i-1}$. This is true also for $Q_i = G_i Q_{i-1}$.

For this reason, we can avoid going through the full G_i and just multiply $\hat{G}$ with the interested rows of $\underline{A}_{i-1}$ and Q_{i-1}, leaving the rest untouched.

An additional observation that can be made is that, when the input matrix is tridiagonal (as in our case, see Section 4), then in $\underline{A}_i$ only a block of size 2×3 on the diagonal is affected by the multiplication by G_i: on the right, there are and there will be all zeros. We can exploit this fact to further improve the algorithm and only compute the rotation for the blocks on the diagonal.

We show the pseudocode of our implementation in Algorithm 5.

Algorithm 5 QR factorization for Hessenberg

```
1:  function HESSENBERG_QR($H$)   ▷ $H$ is an $(n+1) \times n$ augmented Hessenberg matrix
2:      $Q \leftarrow I_{n+1}$
3:      $R \leftarrow H$
4:      for $i = 1$ to $n$ do
5:                          ▷ Compute a rotation with entries at and under the diagonal
6:          $\hat{G} \leftarrow$ ROTATION($R_{i,i}, R_{i+1,i}$)
7:          ▷ Only rows $i$ and $i+1$ change in $R$ and $Q$, so we can update just those rows.
8:          $R(i : i+1, i : i+2) \leftarrow \hat{G} * R(i : i+1, i : i+2)$
9:          $Q(i : i+1, :) \leftarrow \hat{G} * Q(i : i+1, :)$
10:     return $Q^T, R$
```

Cost. Inside the loop, we perform two matrix multiplications of size resp. $(2 \times 2) \times (2 \times 3)$ and $(2 \times 2) \times (2 \times (n+1))$. The first is constant, the second costs $O(n)$. We repeat the loop n times, so the cost of the whole function is $O(n^2)$.

6 Numerical experiments

For testing our algorithm, we wrote a MATLAB function to read graphs in DIMACS format and produce a sparse incidence matrix. For simplicity and repeatability, from the same file we also read information about edge capacity and we use it to populate the diagonal matrix. We also read information about costs and node deficits, and we use these values to populate the other vectors used as the inputs for our algorithm ($\boldsymbol{b}$ and $\boldsymbol{c}$).

Data loading. We use the node deficits information from the DIMACS files to produce the values of the vector $\boldsymbol{c}$ as-is, and we do the same with the costs in order to fill in the vector $\boldsymbol{b}$. As for the diagonal entries of the matrix D, we decided to run a few experiments to evaluate the impact of different initializations. We show in Table 1 the result of these experiments. In particular, we can observe that the convergence time doesn't seem to be affected too much by the different initializations, while the final residual varies a lot. To evaluate convergence time we decided to simplify by using the upper capacity information from the file in order to fill in the diagonal values (note that the sizes exactly match, and also the fact that we have only positive entries). This also allowed us to achieve a better repeatability when running the tests multiple times. However, we should stress that our settings represent a simpler problem with respect to those from actual KKT systems.

Table 1: Results after running the algorithm over `goto16_64` with different generation strategies for $\boldsymbol{d}$. Symbols $\mathcal{U}$, $\mathcal{N}$ and $\mathcal{N}^{-1}$ represent resp. the uniform, Gaussian and inverse Gaussian distributions. The (+) operator indicates a mixture distribution of the operands.

Init. strategy	Time (s)	Rel. time	Iter	$\frac{\|Ax-b\|}{\|b\|}$	cond(A)
upper capacities	1.6816	0.62	16	1.02e−11	5.81e17
ones	2.2310	0.82	33	7.69e−13	1.99e25
$\mathcal{U}(1, 1000)$	2.1645	0.79	39	2.31e−14	4.46e25
$\mathcal{U}(1, 10000)$	2.7274	1.00	52	6.13e−14	2.30e25
$\mathcal{U}(1, 100000)$	2.5698	0.94	57	4.23e−14	6.59e18
$\mathcal{U}(1\text{e}{-9}, 1\text{e}{-7}) + \mathcal{U}(1\text{e}3, 1\text{e}5)$	2.2795	0.84	35	1.09e−3	4.64e17
$\mathcal{N}^{-1}(1\text{e}{-8}, 1\text{e}{-8}) + \mathcal{N}(1\text{e}8, (1\text{e}7)^2)$	2.2751	0.83	34	1.12e−3	4.55e17

All the following tests are related to the resolution of the inner linear system, i.e. the one that we discussed in Section 2, as this is the actual core of the problem.

6.1 Further optimizations.

We used the MATLAB profiler to identify bottlenecks in our algorithm. We identified the following points for optimization:

1. Modifying `hessenberg_qr` to build up on the values of the previous computation. We modified the code in order to output also the last two used rotation matrices, and to take them as a parameter. For this reason, we completely removed the loop, as shown in Algorithm 6.

2. Moving the computation of $\boldsymbol{x}^{(n)}$ in `gmres.m` outside the loop. This is made possible by the fact that we use a formula to compute the residuals that doesn't depend on $\boldsymbol{x}$: see Section 3, Equation 6.

3. Using incomplete Cholesky factorization as a preconditioner (See Section 6.2 for details).

4. Inlining the `arnoldi` function in order to avoid MATLAB making copies of the input matrices.

5. Transforming H_n in a preallocated sparse matrix of size (max_iterations + 1) $\times$ max_iterations.

6. Preallocating columns of Q_n in an exponential way: extending the width of Q_n at each iteration is costly, so we preallocate a given number of empty columns and, at each overflow, we double the number of columns of the matrix.

7. Using the runtime `norm` function instead of manual square roots in the computation of the rotation matrix for `hessenberg_qr`.

We implemented all these optimizations, and this allowed to considerably improve the performance of our algorithm. In Table 2 we show the contributions of some of the most important optimizations that we performed. We tested the performance on a graph with 2^{16} nodes and 524288 edges generated with the "gridgen" generator from `http://www.di.unipi.it/optimize/Data/MCF.html`, in particular we used the `grid16_8_1.dmx` instance.

We used MATLAB's `tic` and `toc` to measure the performance of each configuration 10 times, then we took the minimum. The absolute times are computed on an Intel Core i5–6267U @ 2.90GHz, while the relative times are expressed with respect to the maximum absolute time in the table.

6.2 Incomplete Cholesky factorization as a preconditioner

The incomplete Cholesky factorization produces a matrix L with zero fill-in such that $L^T L \approx A$. We can use this matrix to precondition our problem:

$$A\boldsymbol{x} = \boldsymbol{b}$$

Algorithm 6 QR factorization for Hessenberg – one iteration per call

```
1: function HESSENBERG_QR($H, Q, R, G_1, G_2$)    ▷ $H$ is an $(n+1) \times n$ augmented Hessenberg matrix
2:     $R \leftarrow \begin{bmatrix} R & 0 \\ 0 & 0 \end{bmatrix}$
3:     $Q \leftarrow \begin{bmatrix} Q & 0 \\ 0 & 0 \end{bmatrix}$
4:         ▷ We need to add to $R$ the values from the last column of $H$ that would have been affected if they were already present in the previous iteration. Then we apply to them the rotations from the previous iteration.
5:     $R(n-1:n-1, n) \leftarrow \begin{bmatrix} R & H(n-2:n, end) \end{bmatrix}$
6:     $R(n-2:n-1, n) = G_1 * R(n-2:n-1, n)$
7:     $R(n-1:n, n) = G_2 * R(n-1:n, n)$
8:     ▷ Go on as usual, but only perform iteration $n$ of the loop in the original code.
9:     $\hat{G} \leftarrow$ ROTATION($R_{n,n}, R_{n+1,n}$)
10:    ▷ Only rows $n$ and $n+1$ change in $R$ and $Q$, so we can update just those rows.
11:    $R(n:n+1, n:n+2) \leftarrow \hat{G} * R(n:n+1, n:n+2)$
12:    $Q(n:n+1, :) \leftarrow \hat{G} * Q(n:n+1, :)$
13:                ▷ Shift $Q_1$ and $Q_2$ so that they're available for the next iteration.
14:    $G_1 \leftarrow G_2$
15:    $G_2 \leftarrow \hat{G}$
16:    return $Q^T, R, G_1, G_2$
```

$$A(R^{-1}R)\boldsymbol{x} = \boldsymbol{b}$$
$$A(R^{-1}R)\boldsymbol{x} = \boldsymbol{b}$$
$$R^{-T}A(R^{-1}R)\boldsymbol{x} = R^{-T}\boldsymbol{b}$$

Basically, we can now solve this new linear system instead, and then find x accordingly:

$$\underbrace{R^{-T}AR^{-1}}_{\text{symmetric } \hat{A}}\underbrace{(R\boldsymbol{x})}_{\hat{x}} = \underbrace{R^{-T}\boldsymbol{b}}_{\hat{b}}$$

In our entry point `solvels`, we modify the lambda function like shown in Algorithm 7.

Performance improvement. This optimization brought down the minimum running time on our tests from 1.27 seconds to 0.65 seconds, with the same experimental settings reported in Section 6.1. The number of iterations needed for convergence went down from 223 to 43.

Table 2: Results obtained over a grid graph with 65536 nodes and 524288 edges. "Full optimizations" represents the final version of the code. All the other rows have been measured with the non-preconditioned implementation. "Legacy QR" represents the version of `hessenberg_qr` that started each time from scratch. "Unoptimized residual" indicates the code in `gmres` that computes the residual by multiplying the $z \times z$ matrix A.

	Time (s)	Relative time
Full optimizations	0.563617	0.0411
Full optimizations (non precond.)	1.279131	0.0933
Legacy QR	2.079452	0.1516
Without Lanczos	1.352477	0.0986
Unoptimized residual	13.714996	1.0

Algorithm 7 Entry point for the software

```
1: function SOLVELS($\boldsymbol{d}, E, \boldsymbol{b}, \boldsymbol{c}$, max_iterations, tol)
2:     ...
3:     $L \leftarrow$ ICHOL($\tilde{A}$)
4:     $f \leftarrow \lambda \boldsymbol{v} \,.\, L \setminus (\tilde{A}(L^T \setminus \boldsymbol{v}))$
5:                                              ▷ Solve the linear system to find y
6:     $\boldsymbol{y} \leftarrow L^T \setminus$ GMRES($f, L \setminus \tilde{\boldsymbol{b}}$, max_iterations, tol)
7:     ...
```

6.3 Performance analysis after optimizations.

Given the fact that the linear system that we're actually solving has size $z \times z = |N| \times |N|$, we suspected that the number of nodes whould have had a larger impact on the performance, with respect to the number of edges.

We tried to verify this hypotesis by testing the implementation on a different number of instances with different configurations. In particular, we decided to test three different structures (`goto`, `net` and `grid`) plus a random one. The generators for the first three structures are available from `http://www.di.unipi.it/optimize/Data/MCF.html`.

Our random graph generator works by appropriately filling in an incidence matrix with a predetermined size. We generate node deficits as random numbers from 1 to 10000, such that the second half of the vector is the same as the first part, but with opposite sign (so that the total sum is zero). Linear costs are generated positive and at random from 1 to 10000. The values for $\boldsymbol{d}$ are generated positive and at random from 1 to 4000. Even if this generator is not completely random (look, for example, at the strong relation between node deficits: there always exist two nodes with the same deficit, just with the sign flipped), it helped us to conduct some experiments in a quicker way thanks to the fact that we can dynamically change the generator parameters in the MATLAB code itself (instead of going through the generation of a lot of custom parameters for the existing generators, then generating the files, and finally loading them in MATLAB).

Each instance has a name that consists in a pair of integers that indicate the size of the graph: for `v.e` we have that the number of nodes is 2^v and the number of edges is $2^v e$.

In Table 3 we report the time and iteration number for each of the instances that we tested.

Table 3: Results of testing different instances with different size and structure. Missing entries are nets for which we weren't able to produce results because of limitations in the generator (e.g. it hangs or reports that the problem is too large).

Instance	`goto`		`net`		`grid`		`random`	
	Time	Iter	Time	Iter	Time	Iter	Time	Iter
10.8	0.0130	17	0.0111	30	0.0152	26	0.0090	23
10.32	0.0168	12	0.0221	17	0.0227	20	0.0265	22
10.64	0.0216	10	0.0351	16	0.0321	14	0.0375	20
12.8	0.0287	29	0.0418	40	0.0337	34	0.0384	36
12.64	0.0745	11	0.1438	18	0.1565	17	0.1902	28
12.256	0.2997	8	1.1770	12	1.3277	11	1.4943	21
14.8	0.0845	27	0.1825	57	0.1191	38	0.1617	45
14.64	0.3598	13	0.9316	20	1.0320	18	1.4346	29
16.8	0.3216	26	–	–	0.5806	43	1.2344	72
16.64	1.7189	16	–	–	6.6732	19	9.0780	35

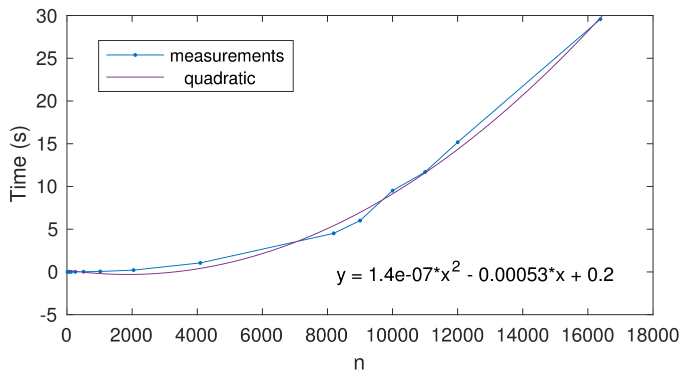
We decided to investigate how the algorithm scales with respect to the number of nodes or the number of edges. We generated various random graphs with different sizes and we plotted running time and number of iterations. As we can see in Figure A.2, for random graphs the algorithm seems to scale linearly with the number of edges, while the number of nodes doesn't contribute as much as we thought. However, by only looking at the time consumed by the `gmres` function, we can see a linear growth over the number of nodes and a sublinear one over the number of edges, at least for the sizes that we tested.

6.4 Experimental cost of hessenberg_qr.

We ran a few experiments to determine the actual scalability of the "legacy" `hessenberg_qr` function. We generated 15 random symmetric Hessenberg matrices with increasing sizes, and we measured the computation time.

As we can see from Figure 1, the function scales as n^2, at least for $n \leq 2^{14}$.

Figure 1: Computation time of the hessenberg_qr function.



6.5 Comparison with existing tools.

MATLAB provides many methods for solving generic linear systems, for example the “backslash” operator and the built-in GMRES or CG function.

We used the same graph and the same metodology illustrated in Section 6.1 to produce the data in Table 4 (non preconditioned) and Table 5 (preconditioned).

Table 4: Comparison with algorithms for solving linear systems from the MATLAB runtime. These results are produced without any manual preconditioning.

	Time (s)	Relative time	Iterations	$\|Ax - b\|$
Our implementation	1.27	0.24	223	1.946 808e−9
Using builtin GMRES	5.33	1.00	219	1.372 565e−9
Using builtin CG	0.89	0.17	233	9.772 200e−10

Table 5: Comparison with algorithms for solving linear systems from the MATLAB runtime. Preconditioned versions.

	Time (s)	Relative time	Iterations	$\|Ax - b\|$
Our implementation	0.56	0.76	43	3.521 168e−10
Using builtin GMRES	0.74	1.00	43	5.714 774e−10
Using builtin CG	0.56	0.76	43	3.695 778e−10

As we can see, knowing the structure of the problem allows evident boosts of performance. But, in particular, preconditioning made our implementation competitive to the Conjugate Gradient from the MATLAB runtime, in terms of convergence speed and residual.

7 Conclusions

We showed the implementation of an optimized solver of a specific class of linear systems coming from flow optimization problems on graphs.

Progressive improvements to the algorithm allowed us to achieve very fast convergence times, which are competitive with MATLAB's built-in methods like GMRES or Conjugate Gradient.

In the analysis section we showed a list of the performance improvements that we did, led by a profiling process. It's interesting to notice, however, that the most fundamental improvements were all based on theoretical ground about linear algebra and optimization: most of the optimizations related to the specific behavior of the language (MATLAB) had a small role in the overall achievements.

References

[1] Youcef Saad and Martin H. Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: `10.1137/0907058`. eprint: `https://doi.org/10.1137/0907058`. URL: `https://doi.org/10.1137/0907058`.

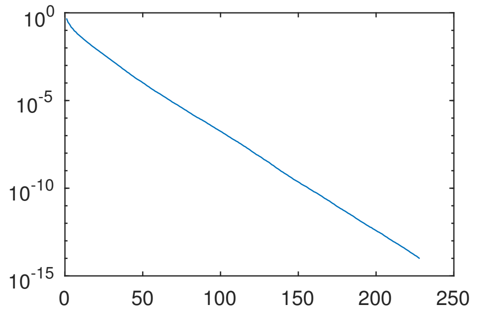
[2] Imad M. Jaimoukha and Ebrahim M. Kasenally. "Krylov Subspace Methods for Solving Large Lyapunov Equations". In: *SIAM Journal on Numerical Analysis* 31.1 (1994), pp. 227–251. DOI: `10.1137/0731012`. eprint: `https://doi.org/10.1137/0731012`. URL: `https://doi.org/10.1137/0731012`.

[3] Youcef Saad. "Practical Use of Some Krylov Subspace Methods for Solving Indefinite and Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 5.1 (1984), pp. 203–228. DOI: `10.1137/0905015`. eprint: `https://doi.org/10.1137/0905015`. URL: `https://doi.org/10.1137/0905015`.

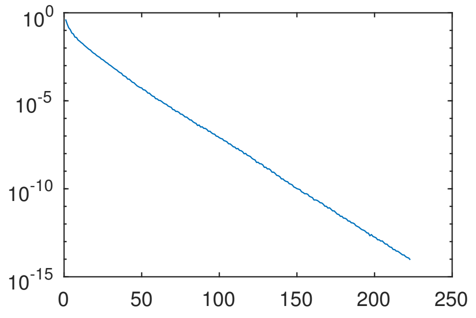
[4] Cornelius Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators". In: *J. Res. Natl. Bur. Stand. B* 45 (1950), pp. 255–282. DOI: `10.6028/jres.045.026`.

A Appendix

Figure A.1: Difference between the two residuals. On the left we have the inefficient computation, and on the right the improved one. We can see that the inefficient one looks slightly more stable.

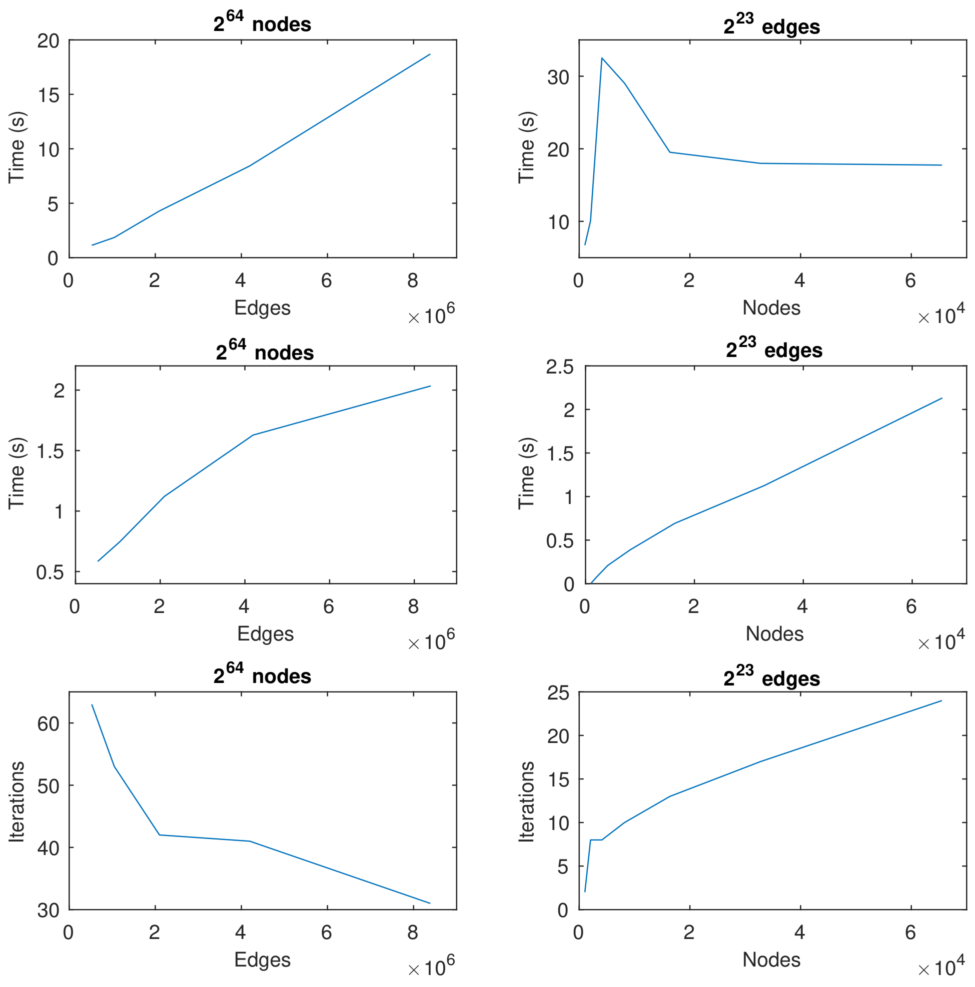


(a) Inefficient computation, logarithmic y



(b) Efficient computation, logarithmic y

Figure A.2: Comparison of running time and iteration number with respect to the number of edges (left column) and the number of nodes (right column). In the first row we measured the time for the entire algorithm, and in the second row one we only show the time consumed by `gmres`.


2^{64} nodes
Time (s)
Edges
$\times 10^6$
2^{23} edges
Time (s)
Nodes
$\times 10^4$
2^{64} nodes
Time (s)
Edges
$\times 10^6$
2^{23} edges
Time (s)
Nodes
$\times 10^4$
2^{64} nodes
Iterations
Edges
$\times 10^6$
2^{23} edges
Iterations
Nodes
$\times 10^4$