NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 3
Linear Least Squares, QR decomposition

P. Bansal

Problem 3.1: Estimating a Tridiagonal Matrix

To determine the least squares solution of an overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ we minimize the residual norm $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$ w.r.t. $\mathbf{x}$. However, we also face a linear least squares problem when minimizing the residual norm w.r.t. the entries of $\mathbf{A}$.
Template: `tridiagleastsquares.cpp`

Let two vectors $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$, for $n > 2 \in \mathbb{N}$, be given. Define α^* and β^* as:

$$(\alpha^*, \beta^*) = \underset{\alpha,\beta\in\mathbb{R}}{\operatorname{argmin}} \left\| \mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} \right\|_2, \tag{3.1}$$

where $\mathbf{T}_{\alpha,\beta} \in \mathbb{R}^{n\times n}$ is the following tridiagonal matrix:

$$\mathbf{T}_{\alpha,\beta} = \begin{bmatrix} \alpha & \beta & 0 & \dots & 0 \\ \beta & \alpha & \beta & \ddots & \vdots \\ 0 & \beta & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \dots & 0 & \beta & \alpha \end{bmatrix} \tag{3.2}$$

(a) Reformulate Eq. (3.1) as a linear least squares problem:

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^k}{\operatorname{argmin}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2, \tag{3.3}$$

where $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{b} \in \mathbb{R}^m$, for $m, k \in \mathbb{N}$.

Hint: For $\mathbf{x} = [\alpha, \beta]^\top$, find $\mathbf{A}$ such that $\mathbf{T}_{\alpha,\beta}\mathbf{z} = \mathbf{A}\mathbf{x}$.

SOLUTION:

The vector $\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c}$ whose norm has to be minimized in Eq. (3.1) can be written as:

$$\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} = \begin{bmatrix} \alpha z_1 + \beta z_2 \\ \alpha z_2 + \beta(z_1 + z_3) \\ \dots \\ \alpha z_{n-1} + \beta(z_{n-2} + z_n) \\ \alpha z_n + \beta z_{n-1} \end{bmatrix} - \mathbf{c} = \begin{bmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - \mathbf{c} =: \mathbf{A}\mathbf{x} - \mathbf{b} \tag{3.4}$$

In this form, we have moved the unknown α and β into vector $\mathbf{x}$, while the data $\mathbf{z}$ and $\mathbf{c}$ are moved into matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{b} \equiv \mathbf{c}$ (which is not affected by the transformation). The number of unknown k is equal to 2 and the number of equations m is equal to n.

(b) Implement a C++ function which solves the linear least squares problem of Eq. (3.1) using the normal equation method and returns the optimal parameters α^* and β^*:

```
VectorXd lsqEst(const VectorXd &z, const VectorXd &c);
```

SOLUTION:

C++11-code 3.1: Solution of (b)

```
{
    // Initialization
    int n = z.size();
    assert( z.size() == c.size() && "z and c must have same size");

    VectorXd x(2);

    MatrixXd A(n,2);
    A.col(0) = z;
    A(0,1) = z(1);
    for(size_t i=1; i<n-1; ++i) {
        A(i,1) = z(i-1) + z(i+1);
    }
    A(n-1,1) = z(n-2);

    // Normal equations
    MatrixXd lhs = A.transpose() * A; // Left-hand side
    VectorXd rhs = A.transpose() * c; // Right-hand side
    x = lhs.fullPivLu().solve(rhs);

    return x;
}
```

Let $\mathbf{X} \in \mathbb{R}^{m,n}$, $\mathbf{a} \in \mathbb{R}^m$ and $\mathbf{b} \in \mathbb{R}^n$. Consider the scalar functions:

$$\Phi_1(\mathbf{X}) = \|\mathbf{X}\|_F^2 \tag{3.5a}$$
$$\Phi_2(\mathbf{X}) = \mathbf{a}^\top \mathbf{X}\mathbf{b}. \tag{3.5b}$$

Here $\|\cdot\|_F$ is the Frobenius norm of a matrix.

(c) Compute $\dfrac{\partial(\Phi_1(\mathbf{X}))}{\partial\mathbf{X}}$ and $\dfrac{\partial(\Phi_2(\mathbf{X}))}{\partial\mathbf{X}}$.

SOLUTION:

$$\frac{\partial(\Phi_1(\mathbf{X}))}{\partial\mathbf{X}} = 2\mathbf{X} \tag{3.6a}$$
$$\frac{\partial(\Phi_2(\mathbf{X}))}{\partial\mathbf{X}} = \mathbf{a}\mathbf{b}^\top \tag{3.6b}$$

Problem 3.2: Sparse Approximate Inverse (SPAI)

The SPAI method is a technique used in the numerical solution of partial differential equations. From a least squares viewpoint, we encounter a non-standard least squares problems. SPAI techniques are applied to huge and extremely sparse matrices, say, of dimension $10^7 \times 10^7$ with only 10^8 non-zero entries. Therefore, sparse matrix techniques must be applied.

Let $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, be a regular sparse matrix with at most $n \ll N$ non-zero entries per row and column. We define the space of matrices with the same pattern as $\mathbf{A}$:

$$\mathcal{P}(\mathbf{A}) := \{\mathbf{X} \in \mathbb{R}^{N,N} : (\mathbf{A})_{ij} = 0 \Rightarrow (\mathbf{X})_{ij} = 0\}. \tag{3.7}$$

The "primitive" SPAI (sparse approximate inverse) $\mathbf{B}$ of $\mathbf{A}$ is defined as

$$\mathbf{B} := \underset{\mathbf{X} \in \mathcal{P}(\mathbf{A})}{\operatorname{argmin}} \|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F , \tag{3.8}$$

where $\|\cdot\|_F$ stands for the Frobenius norm.

(a) Show that the columns of $\mathbf{B}$ can be computed independently of each other by solving linear least squares problems. Denote columns of $\mathbf{B}$ by $\mathbf{b}_i$.

SOLUTION:

Let $\mathbf{x}_i, \mathbf{b}_i, \mathbf{a}_i$ denote the i-th columns of the matrices $\mathbf{X}, \mathbf{B}, \mathbf{A}$ in (3.8), respectively. Notice, that the values in the i-th column of $\mathbf{I} - \mathbf{A}\mathbf{X}$ depend only on the values in the vector $\mathbf{x}_i$. Hence, in order to minimize the Frobenius norm $\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2$ one needs to (independently) minimize each of the terms $\|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2$:

$$\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{I}\mathbf{e}_i - \mathbf{A}\mathbf{X}\mathbf{e}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2. \tag{3.9}$$

The corresponding minimization problems read

$$\mathbf{b}_i := \underset{\mathbf{x}_i \in \mathcal{P}(\mathbf{a}_i)}{\operatorname{argmin}} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \dots, N, \tag{3.10}$$

where for a vector $\mathbf{a}$ we define $\mathcal{P}(\mathbf{a}) := \{\mathbf{x} \in \mathbb{R}^N : a_i = 0 \Rightarrow x_i = 0\}$.

(b) Implement an efficient C++ function

```
SparseMatrix<double> spai(SparseMatrix<double> & A);
```

for the computation of $\mathbf{B}$ according to (3.8). You may rely on the normal equations associated with the linear least squares problems for computing the columns of $\mathbf{B}$ or you may simply invoke the least squares solver of EIGEN.

Hint: Exploit the underlying CCS data structure of `SparseMatrix<double>` Moreover, build the output matrix $\mathbf{B}$ in EIGEN using the intermediate triplet format.

SOLUTION:

The formula (3.10) is not yet in the canonical form. Fix i. Define $m_i := nnz(\mathbf{a}_i)$. Notice that $\mathcal{P}(\mathbf{a}_i) \simeq \mathbb{R}^{m_i}$ via the mapping $\mathbf{x} \mapsto [x_{j_1}, \dots, x_{j_{m_i}}] =: \tilde{\mathbf{x}}$, where j_k, $k = 1, \dots, m_i$ are the indices s.t. $a_{j_k} \neq 0$.

The product $\mathbf{Ax}$ can be expressed, equivalently, with $\mathbf{C}\tilde{\mathbf{x}}$, where $\mathbf{C} \in \mathbb{R}^{N \times m_i}$ is defined as the matrix $\mathbf{A}$, without the columns i, s.t. $a_i = 0$:

$$\mathbf{C} := [\mathbf{A}_{j_1}, \dots, \mathbf{A}_{j_{m_i}}] .$$

We obtain:

$$\mathbf{b}_i := \underset{\mathbf{x}_i \in \mathbb{R}^{m_i}}{\operatorname{argmin}} \|\mathbf{e}_i - \mathbf{C}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \dots, N. \tag{3.11}$$

This canonical form can be solved with the standard normal formula.

C++11-code 3.2: Computation of $\mathbf{B}$.

```cpp
SparseMatrix<double> spai(SparseMatrix<double> & A) {
    // Size check
    assert(A.rows() == A.cols() &&
    "Matrix must be square!");
    unsigned int N = A.rows();

    A.makeCompressed();

    // Obtain pointers to data of A
    double* valPtr = A.valuePtr();
    index_t* innPtr = A.innerIndexPtr();
    index_t* outPtr = A.outerIndexPtr();

    // Create vector for triplets of B and reserve enough space
    std::vector<Triplet<double>> B_triplets;
    B_triplets.reserve(A.nonZeros());

    // Project P(A) onto P(a_i) and compute b_i
    for(unsigned int i = 0; i < N; ++i) {
        // Number of non-zeros in a_i
        index_t nnz_i = outPtr[i+1] - outPtr[i];
        if(nnz_i == 0) continue; // skip column, if empty

        // Smaller and denser matrix to store
        // non-zero elements relevant for computing b_i
        SparseMatrix<double> C(N, nnz_i);
        std::vector<Triplet<double>> C_triplets;
        C_triplets.reserve(nnz_i*nnz_i);

        // Build matrix C.
        for(unsigned int k = outPtr[i]; k < outPtr[i+1]; ++k) {
            // Row index of non-zero element in column b_i
            index_t row_k = innPtr[k];
            // Number of non-zero entries in (row_k)-th column
            index_t nnz_k = outPtr[row_k+1] - outPtr[row_k];
            // Loop over all non-zeros of row_kth-column
            // Store the triplet for the non-zero element
            for(unsigned int l = 0; l < nnz_k; ++l) {
                unsigned int innIdx = outPtr[row_k] + l;
                C_triplets.emplace_back(Triplet<double>(innPtr[innIdx], k
                    - outPtr[i], valPtr[innIdx]));
            }
        }
```

```
        C.setFromTriplets(C_triplets.begin(), C_triplets.end());
        C.makeCompressed();

        // Normal equation method: b_i = (C^T C)^{-1} C^T e_i
        SparseMatrix<double> S = C.transpose() * C;
        VectorXd xt = C.row(i).transpose();
        SparseLU<SparseMatrix<double>> spLU(S);
        VectorXd b = spLU.solve(xt);

        // store the triplets for elements b_i
        for(unsigned int k = 0; k < b.size(); ++k) {
            B_triplets.emplace_back(Triplet<double>(innPtr[outPtr[i] + k],
               i, b(k)));
        }
    }

    // Build and return B
    SparseMatrix<double> B = SparseMatrix<double>(N,N);
    B.setFromTriplets(B_triplets.begin(), B_triplets.end());
    B.makeCompressed();
    return B;
}
```

(c) What is the total asymptotic computational effort of `spai` in terms of the problem size parameters N and n?

SOLUTION:

In the body of the `for` loop (of size N) the matrix $\mathbf{C}$ is assembled with two for loops of length at most n, then the matrix $\mathbf{C}^\top\mathbf{C}$ is assembled with complexity $O(n^3)$ (note: $\mathbf{C}$ is sparse, with only n non-zeros per column), and then the corresponding linear system of size $n \times n$ is solved (complexity $O(n^3)$). Hence, the resulting total complexity is dominated by $O(Nn^3)$.

Problem 3.3: QR decomposition

In this problem, we study the QR decomposition computed via Cholesky decomposition and Householder reflections. Refer section (2.8.13) in the lecture notes to read about Cholesky decomposition.
Template: `choleskyQR.cpp`

(a) Given a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, s.t. $\operatorname{rank}(\mathbf{A}) = n$, show that $\mathbf{A}^\top\mathbf{A}$ admits a Cholesky decomposition.

Hint: Cholesky decomposition of a matrix $\mathbf{B}$ exists only if $\mathbf{B}$ is symmetric and positive definite.

SOLUTION:

To prove: $\mathbf{A}^\top\mathbf{A}$ is symmetric positive definite.

Proof:

$(\mathbf{A}^\top\mathbf{A})^\top = \mathbf{A}^\top(\mathbf{A}^\top)^\top = \mathbf{A}^\top\mathbf{A} \implies$ symmetric.

$\operatorname{rank}(\mathbf{A}) = n \implies \mathbf{A}$ is injective. Hence, $\forall \mathbf{v} \in \mathbb{R}^n \mid \mathbf{v} \neq \mathbf{0}$, it holds that $\mathbf{A}\mathbf{v} \neq \mathbf{0}$.

$\implies \mathbf{v}^\top\mathbf{A}^\top\mathbf{A}\mathbf{v} = |\mathbf{A}\mathbf{v}|_2^2 > 0$, which means that $\mathbf{A}^\top\mathbf{A}$ is positive definite.

(b) Implement an EIGEN based C++ function:

```
void CholeskyQR(const MatrixXd & A, MatrixXd & Q, MatrixXd & R);
```

which computes an *economical* QR decomposition of a given full-rank matrix `A`. Give an analytical proof that your `CholeskyQR` works.

SOLUTION:

C++11-code 3.3: QR-decomposition via Cholesky decomposition

```
void CholeskyQR(const MatrixXd & A, MatrixXd & R, MatrixXd & Q) {

    MatrixXd AtA = A.transpose() * A;
    LLT<MatrixXd> chol = AtA.llt();
    MatrixXd L = chol.matrixL();
    R = L.transpose();
    Q = L.triangularView<Lower>().solve(A.transpose()).transpose();

}
```

We need to verify three properties:

- $\mathbf{R}$ is upper triangular.
- $\mathbf{Q}$ is orthogonal.
- $\mathbf{QR} = \mathbf{A}$

This is done in the following:

- $\mathbf{R}$ is upper triangular from definition of Cholesky decomposition.

- Given that $\mathbf{Q} = \left(\mathbf{R}^{-\top}\mathbf{A}^\top\right)^\top = \mathbf{A}\mathbf{R}^{-1}$ and, from the Cholesky decomposition, $\mathbf{R}^\top\mathbf{R} = \mathbf{A}^\top\mathbf{A} \Rightarrow \mathbf{R}^{-\top}\mathbf{A}^\top\mathbf{A}\mathbf{R}^{-1} = \mathbf{I}$, we have that $\mathbf{Q}^\top\mathbf{Q} = \mathbf{R}^{-\top}\mathbf{A}^\top\mathbf{A}\mathbf{R}^{-1} = \mathbf{I}$. This means that $\mathbf{Q}$ is orthogonal.
- $\mathbf{Q}\mathbf{R} = \left(\mathbf{R}^{-\top}\mathbf{A}^\top\right)^\top\mathbf{R} = \mathbf{A}\mathbf{R}^{-1}\mathbf{R} = \mathbf{A}$

(c) Implement an EIGEN based C++ function:

```
void DirectQR(const MatrixXd & A, MatrixXd & Q, MatrixXd & R);
```

which computes an *economical* QR decomposition of `A` using `HouseholderQR` class from EIGEN. Compare the results from `CholeskyQR` and `DirectQR`.

SOLUTION:

C++11-code 3.4: QR-decomposition via Householder reflections in EIGEN

```cpp
void DirectQR(const MatrixXd & A, MatrixXd & R, MatrixXd & Q) {

    size_t m = A.rows();
    size_t n = A.cols();

    HouseholderQR<MatrixXd> QR = A.householderQr();
    Q = QR.householderQ() * MatrixXd::Identity(m, std::min(m, n));
    R = MatrixXd::Identity(std::min(m, n), m) *
        QR.matrixQR().triangularView<Upper>();
    // If A: m x n, then Q: m x m and R: m x n.
    // If m > n, however, the extra columns of Q and extra rows of R
       are not needed: "full" to "economical" QR

}
```

(d) Let EPS denote the machine precision. Does your function `CholeskyQR` return the correct result, compare with `DirectQR`, for $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \frac{1}{2}\mathrm{EPS} & 0 \\ 0 & \frac{1}{2}\mathrm{EPS} \end{bmatrix}$? Explain.

SOLUTION:

$\mathbf{A}$ has rank 2, but $\mathbf{A}^\top\mathbf{A} = \begin{bmatrix} 1+\frac{1}{4}\epsilon^2 & 1 \\ 1 & 1+\frac{1}{4}\epsilon^2 \end{bmatrix} \approx \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ has rank 1 in machine arithmetic. Hence, $\mathbf{A}^\top\mathbf{A}$ is symmetric but not positive definite: the hypothesis needed for the Cholesky decomposition are not satisfied.

Problem 3.4: Givens rotations

In this problem, we look at Givens rotations to compute a QR decomposition and also briefly compare it with Householder reflections.

Let $\mathbf{A} \in \mathbb{R}^{3,2}$,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ \sqrt{2} & 1 \end{bmatrix} \tag{3.12}$$

(a) Use Householder reflections to transform $\mathbf{A}$ to an upper triangular matrix $\tilde{\mathbf{A}}$. Perform the computations and show the steps on paper.

SOLUTION:

$$\begin{aligned}
\mathbf{a}_1 &= \begin{bmatrix}1 & 1 & \sqrt{2}\end{bmatrix}^\top \implies \|\mathbf{a}_1\|_2 = 2 \\
\mathbf{v}_1 &= \frac{1}{2}(\mathbf{a}_1 - \|\mathbf{a}_1\|_2\mathbf{e}_1) \implies \mathbf{v}_1 = \frac{1}{2}\begin{bmatrix}-1 & 1 & \sqrt{2}\end{bmatrix}^\top \\
\mathbf{H}_1(\mathbf{V}_1) = \mathbf{I} - 2\frac{\mathbf{v}_1\mathbf{v}_1^\top}{\mathbf{v}_1^\top\mathbf{v}_1} &= \frac{1}{2}\begin{bmatrix} 1 & 1 & \sqrt{2} \\ 1 & 1 & -\sqrt{2} \\ \sqrt{2} & -\sqrt{2} & 0 \end{bmatrix} \\
\tilde{\mathbf{A}}_1 := \mathbf{H}_1\mathbf{A} &= \frac{1}{2}\begin{bmatrix} 4 & 3+\sqrt{2} \\ 0 & 3-\sqrt{2} \\ 0 & \sqrt{2} \end{bmatrix} \\
\mathbf{a}_2 &= \frac{1}{2}\begin{bmatrix}0 & 3-\sqrt{2} & \sqrt{2}\end{bmatrix}^\top \implies \|\mathbf{a}_2\|_2 = \frac{\sqrt{13-6\sqrt{2}}}{2} \\
\tilde{\mathbf{A}} &= \begin{bmatrix} 2 & (3+\sqrt{2})/2 \\ 0 & \|\mathbf{a}_2\|_2 \\ 0 & 0 \end{bmatrix}
\end{aligned}$$

(b) Verify your computations for Sub-problem (a) using the `HouseholderQR` class provided by EIGEN.

(c) Implement an EIGEN C++ function:

```
void rotInPlane(const Vector2d& x, Matrix2d& G, Vector2d& y);
```

which applies Givens rotation on a 2d vector `x`. It should also avoid cancellation.

SOLUTION:

C++11-code 3.5: Givens rotation of a 2d vector

```
void rotInPlane(const Vector2d& x, Matrix2d& G, Vector2d& y) {

    if (x(1) != 0.0) {

        double t, s, c;
        // to avoid cancellation
        if (std::abs(x(1)) > std::abs(x(0))) {
```

```
            t = x(0)/x(1);
            s = 1./std::sqrt(1 + t*t);
            c = s*t;
        } else {
            t = x(1)/x(0);
            c = 1./std::sqrt(1 + t*t);
            s = c*t;
        }

        G << c,s,-s,c; // 2×2 Givens rotation matrix

    } else G.setIdentity();
    y << x.norm(), 0; // y = Gx

}
```

(d) Implement an EIGEN C++ function:

`void givensQR(const Matrixxd& A, MatrixXd& Q, MatrixXd& R);`

which uses the Givens rotation routine `rotInPlane` successively to compute the QR decomposition of a matrix `A`.

SOLUTION:

C++11-code 3.6: QR decomposition using Givens rotations

```
void givensQR(const MatrixXd & A, MatrixXd & Q, MatrixXd & R) {

    unsigned int m = A.rows();
    unsigned int n = A.cols();
    Q.setIdentity();
    R = A;

    Vector2d x, y;
    Matrix2d G;

    for (int j=0; j<n; j++)
        for (int i=m-1; i>j; i--) {
            x(0) = R(i-1,j);
            x(1) = R(i,j);
            rotInPlane(x, G, y);
            R.block(i-1,j,2,n-j) = G*R.block(i-1,j,2,n-j);
            Q.block(0,i-1,m,2) = Q.block(0,i-1,m,2)*G.transpose();
        }

}
```

(e) Run basic sanity checks for your implementation of `givensQR` and compare your results with that of `HouseholderQR`. Is the QR decomposition unique?

(f) Compare the complexity of `givensQR` and `HouseholderQR` for a general input matrix $\mathbf{A} \in \mathbb{R}^{m,n}$.

SOLUTION:

For Householder reflections, matrix blocks are updated as

$$\mathbf{A}_{k:m,k:n} = \mathbf{A}_{k:m,k:n} - 2\mathbf{v}_k(\mathbf{v}_k^\top \mathbf{A}_{k:m,k:n})$$

for $k = 1, 2, \ldots, n$. The operations involved are:

- $2(m-k)(n-k)$ dot products $\mathbf{v}_k^\top \mathbf{A}_{k:m,k:n}$
- $(m-k)(n-k)$ scalar-vector product $\mathbf{v}^\top(\ldots)$
- $(m-k)(n-k)$ subtraction $\mathbf{A}_{k:m,k:n} - \ldots$

Total ops = $\sum_{k=1}^{n} 4(m-k)(n-k) \sim 2mn^2 - 2n^3/3$

In Givens rotations, a 2×2 matrix $\mathbf{G}$ updates successive pairs of "2-row broad" block of length $n-k$, for $k = 1, 2, \ldots, n$. Therefore, the total number of operations are $\sum_{k=1}^{n} 6(m-k)(n-k) \sim 3mn^2 - n^3$.

Givens rotations are roughly 50% more expensive than Householder reflections. This observation becomes crucial with dealing with large matrices.
