Methods in Computational Science - Direct methods for systems of linear equations (ch.5)

Johan Hoffman

System of linear equations

A system of n linear equations with n unknowns corresponds to the matrix equation

$$Ax = b,$$

where $A \in R^{n \times n}$ is a given matrix, $b \in R^n$ a given vector, and $x \in R^n$ the unknown solution vector. We recall that the vector b can be interpreted as the image of x under the linear transformation A, or alternatively x can be interpreted as the coordinates of b in the column space of A. If A is nonsingular, the solution x can be expressed in terms of the inverse matrix A^{-1} as

$$x = A^{-1}b.$$

Diagonal and orthogonal matrices

For some matrices the inverse matrix is easy to construct, as in the case of a *diagonal matrix* $D = (d_{ij})$, where $d_{ij} = 0$ for all $i \neq j$. Here the inverse is equal to another diagonal matrix $D^{-1} = (d_{ij}^{-1})$, with $d_{ij}^{-1} = 0$ for all $i \neq j$, and

$$d_{ij}^{-1} = (d_{ij})^{-1}, \quad \forall i = j.$$

For an orthogonal matrix $Q = (q_{ij})$ the inverse is equal to the transpose $Q^{-1} = Q^T$, that is,

$$Q^{-1} = (q_{ji}).$$

Triangular matrices

Apart from diagonal and orthogonal matrices, also triangular matrices are easy to invert. We distinguish between two types of triangular matrices; a *lower triangular matrix*

$$L = (l_{ij}), \quad l_{ij} = 0, \forall i < j,$$

and an *upper triangular matrix*

$$U = (u_{ij}), \quad u_{ij} = 0, \forall i > j.$$

Forward and backward substitution

$$Lx = b, \quad Ux = b,$$

or in component form

$$\begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{12} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{2n} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Forward and backward substitution

ALGORITHM 5.1. x = forward_substitution(L, b).
Input: a lower triangular matrix **L** and a vector **b**.
Output: the solution **x** to the equation Lx = b.

```
1:  n = length(x)
2:  x[0] = b[0]/L[0,0]
3:  for i=1:n-1 do
4:      sum = 0
5:      for j=0:i-1 do
6:          sum = sum + L[i,j]*x[j]
7:      end for
8:      x[i] = (b[i] - sum)/L[i,i]
9:  end for
10: return x
```

ALGORITHM 5.2. x = backward_substitution(U, b).
Input: an upper triangular matrix **U** and a vector **b**.
Output: the solution **x** to the equation Ux = b.

```
1:  n = length(x)
2:  x[n-1] = b[n-1]/U[n-1,n-1]
3:  for i=n-2:0 do
4:      sum = 0
5:      for j=i+1:n-1 do
6:          sum = sum + U[i,j]*x[j]
7:      end for
8:      x[i] = (b[i] - sum)/U[i,i]
9:  end for
10: return x
```

$$f(n) = 3 + \sum_{i=1}^{n-1}(4+3i) = 3 + 4(n-1) + 3\frac{n(n-1)}{2} = \frac{3}{2}n^2 + \frac{5}{2}n - 5, \quad f(n) = \mathcal{O}(n^2)$$

Direct method to solve equation $Ax = b$

In conclusion, if we can factorize a general nonsingular matrix into $A = BC$, where B and C are diagonal, orthogonal or triangular matrices, we can solve the system $Ax = b$ as follows,

$$Ax = b \Leftrightarrow BCx = b \Leftrightarrow Cx = B^{-1}b \Leftrightarrow x = C^{-1}B^{-1}b$$

- How do we factorize a general matrix A?

Gram-Schmidt orthogonalization: $A = QR$

For a nonsingular matrix $A \in R^{n \times n}$ the successive vector spaces spanned by its column vectors $a_{:j}$ are denoted by

$$\langle a_{:1} \rangle \subseteq \langle a_{:1}, a_{:2} \rangle \subseteq \langle a_{:1}, a_{:2}, a_{:3} \rangle \subseteq \ldots \subseteq \langle a_{:1}, \ldots, a_{:n} \rangle = \text{range}(A).$$

Since A has full rank, for each such vector space we can construct an orthonormal basis $\{q_i\}_{i=1}^{j}$ such that $\langle q_1, \ldots, q_j \rangle = \langle a_{:1}, \ldots, a_{:j} \rangle$ for all $j \leq n$, that is, $\{q_i\}_{i=1}^{j}$ and $\{a_{:i}\}_{i=1}^{j}$ both span the same vector space. Given the basis vectors $a_{:j}$, we successively construct residual vectors v_j that are orthogonal to the space $\langle q_1, \ldots, q_{j-1} \rangle$ by subtracting the projection of $a_{:j}$ onto the space,

$$v_j = a_{:j} - \sum_{i=1}^{j-1} (a_{:j}, q_i) q_i. \tag{5.4}$$

Gram-Schmidt orthogonalization: $A = QR$

With the notation $r_{ij} = (a_{:j}, q_i)$ and $r_{jj} = \|v_j\|$ we can rewrite equation (5.4) as

$$\begin{aligned} a_{:1} &= r_{11} q_1 \\ a_{:2} &= r_{12} q_1 + r_{22} q_2 \\ &\vdots \\ a_{:n} &= r_{1n} q_1 + \ldots + r_{nn} q_n \end{aligned}$$

which corresponds to the QR factorization $A = QR$, with $q_j = q_{:j}$ the column vectors of Q,

$$\left[\begin{array}{c|c|c|c} & & & \\ a_{:1} & a_{:2} & \cdots & a_{:n} \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ q_{:1} & q_{:2} & \cdots & q_{:n} \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & r_{nn} \end{bmatrix}$$

Gram-Schmidt orthogonalization: $A = QR$

If we let $\hat{Q}_{j-1}$ denote the $n \times (j-1)$ matrix with column vectors $q_{:i}$ for $i \leq j-1$,

$$v_j = a_{:j} - \sum_{i=1}^{j-1} (a_{:j}, q_{:i}) q_{:i} = a_{:j} - \sum_{i=1}^{j-1} q_{:i} q_{:i}^T a_{:j} = (I - \hat{Q}_{j-1} \hat{Q}_{j-1}^T) a_{:j},$$

where $\hat{Q}_{j-1} \hat{Q}_{j-1}^T$ is an orthogonal projector onto range$(\hat{Q}_{j-1})$, which follows from

$$\hat{Q}_{j-1} \hat{Q}_{j-1}^T (\hat{Q}_{j-1} \hat{Q}_{j-1}^T) = \hat{Q}_{j-1} (\hat{Q}_{j-1}^T \hat{Q}_{j-1}) \hat{Q}_{j-1}^T = \hat{Q}_{j-1} \hat{Q}_{j-1}^T,$$

since $\hat{Q}_{j-1}^T \hat{Q}_{j-1}$ is a $(j-1) \times (j-1)$ identity matrix. We then define

$$P_j = I - \hat{Q}_{j-1} \hat{Q}_{j-1}^T$$

to be the complementary projector onto range$(\hat{Q}_{j-1})^{\perp}$, with $P_1 = I$ the identity matrix.

Modified Gram-Schmidt orthogonalization

Classical Gram-Schmidt orthogonalization:

$$q_{:j} = P_j a_{:j} / \|P_j a_{:j}\|, \quad j = 1, ..., n. \qquad P_j = I - \hat{Q}_{j-1} \hat{Q}_{j-1}^T$$

Alternatively, we can construct the projector P_j as the recursive multiplication

$$P_j = P^{\perp q_{:j-1}} \cdots P^{\perp q_{:2}} P^{\perp q_{:1}},$$

with projectors onto subspaces orthogonal to each individual orthonormal basis vector $q_{:i}$,

$$P^{\perp q_{:i}} = I - q_{:i} q_{:i}^T.$$

This modified Gram-Schmidt orthogonalization is more stable in finite precision, since it avoids accumulation of round-off errors.

Modified Gram-Schmidt orthogonalization

ALGORITHM 5.3. (Q, R) = modified_gram_schmidt_iteration(A).
Input: a full rank nxn matrix **A**.
Output: an orthogonal nxn matrix **Q** and an upper triangular nxn matrix **R**.

```
1:  for j=0:n-1 do
2:      v[:] = A[:,j]
3:      for i=0:j-1 do
4:          R[i,j] = scalar_product(Q[:,i], v[:])
5:          v[:] = v[:] - R[i,j]*Q[:,i]
6:      end for
7:      R[j,j] = norm(v)
8:      Q[:,j] = v[:]/R[j,j]
9:  end for
10: return  Q, R
```

Modified Gram-Schmidt orthogonalization

Example 5.1. We now apply Algorithm 5.3 to the symmetric positive definite matrix

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}.$$

If we initialize Q and R as zero matrices $Q = R = 0 \in R^{2\times 2}$, after the first iteration we have

$$Q = \frac{1}{\sqrt{5}} \begin{bmatrix} 2 & 0 \\ -1 & 0 \end{bmatrix}, \quad R = \frac{1}{\sqrt{5}} \begin{bmatrix} 5 & -4 \\ 0 & 0 \end{bmatrix},$$

and after the second iteration

$$Q = \frac{1}{\sqrt{5}} \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix}, \quad R = \frac{1}{\sqrt{5}} \begin{bmatrix} 5 & -4 \\ 0 & 3 \end{bmatrix},$$

from which we can verify that Q is orthogonal, R is upper triangular, and that $QR = A$.

Householder QR factorization

$$Q_{n-1}...Q_2Q_1A = R,$$

and we recall that also the matrix $Q = (Q_{n-1}...Q_2Q_1)^{-1}$ is orthogonal. In the *Householder algorithm*, matrices are chosen of the form

$$Q_k = \begin{bmatrix} I_k & 0 \\ 0 & F_k \end{bmatrix}, \tag{5.10}$$

with I_k the $(k-1)\times(k-1)$ identity matrix, and with F_k an $(n-k+1)\times(n-k+1)$ orthogonal matrix. Q_k is constructed to successively introduce $n-k$ zeros below the diagonal in the kth column of A, while leaving the upper $k-1$ rows untouched, and therefore

$$Q_k\hat{A}_{k-1} = \begin{bmatrix} I_k & 0 \\ 0 & F_k \end{bmatrix}\begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ 0 & \hat{A}_{22} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ 0 & F_k\hat{A}_{22} \end{bmatrix},$$

with

$$\hat{A}_{k-1} = Q_{k-1}\cdots Q_2Q_1A.$$

Householder QR factorization

$$F_k x = \begin{bmatrix} \pm\|x\| \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \pm\|x\| \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \pm\|x\| e_1.$$

Further, we need F_k to be an orthogonal matrix, which we achieve by constructing F_k in the form of a reflector, so that $F_k x$ is the reflection of x in a hyperplane defined by a normal vector

$$v = x - s\|x\|e_1,$$

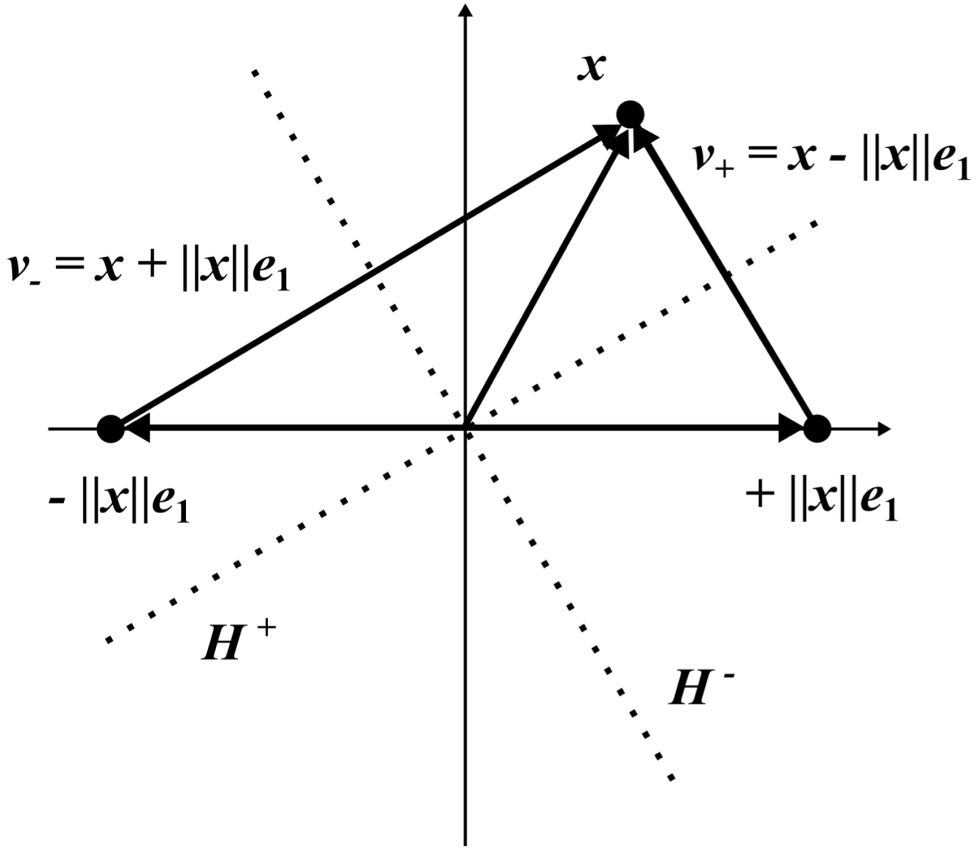
where s denotes the sign $\pm$, that is,

$$F_k = I - 2\frac{vv^T}{v^T v}.$$

Householder QR factorization

$$v = x - s\|x\|e_1,$$

$$F_k = I - 2\frac{vv^T}{v^T v}.$$



Householder QR factorization

To verify that all subdiagonal components are reduced to zero, note that

$$v^T v = (x - s\|x\|e_1)^T (x - s\|x\|e_1) = 2\|x\|(\|x\| - sx_1),$$

and

$$v^T x = (x - s\|x\|e_1)^T x = \|x\|^2 - s\|x\|x_1 = \|x\|(\|x\| - sx_1),$$

so that the reflection $F_k x$ is a scaled Cartesian basis vector,

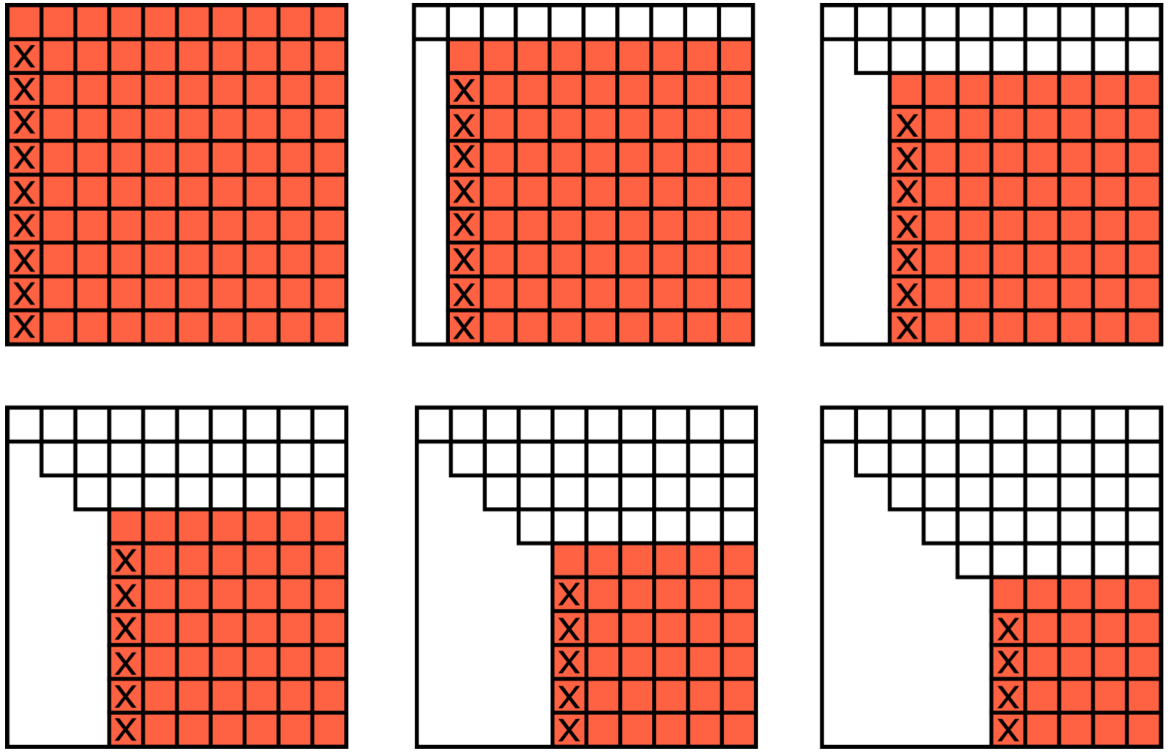
$$F_k x = x - 2\frac{vv^T}{v^T v}x = x - 2\frac{v^T x}{v^T v}v = x - v = s\|x\|e_1.$$

Householder QR factorization

ALGORITHM 5.4. (A,v_0,..,v_n-1) = householder_qr_factorization(A).
Input: a full rank nxn matrix **A**.
Output: upper triangular matrix **A**, n-1 vectors **v_0,..,v_n-2** of lengths n to 2.

```
1:  for k=0:n-2 do
2:      x[:] = A[k:n-1,k]
3:      v_k[:] = x[:]
4:      v_k[0] = v_k[0] - sign(x[0])*norm(x)
5:      v_k[:] = v_k[:]/norm(v_k)
6:      for m=k:n-1 do
7:          A[k:n-1,m] = A[k:n-1,m] - 2*v_k[:]*dot(v_k, A[k:n-1,m])
8:      end for
9:  end for
10: return  A, v_0,..,v_n-2
```

Householder QR factorization



Householder QR factorization

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \qquad v = x - \text{sgn}(x_1)\|x\|e_1 = (2 - \sqrt{5}, -1)^T,$$

so that $v^T v = \|v\|^2 = 10 - 4\sqrt{5}$. The upper triangular matrix R corresponds to the reflections of the column vectors $a_{:1}$ and $a_{:2}$ in the hyperplane H, given by

$$Fa_{:1} = \left(I - 2\frac{vv^T}{v^T v}\right) a_{:1} = a_{:1} - 2\frac{x^T a_{:1}}{v^T v} v = \frac{1}{\sqrt{5}} \begin{bmatrix} 5 \\ 0 \end{bmatrix},$$

$$Fa_{:2} = \left(I - 2\frac{vv^T}{v^T v}\right) a_{:2} = a_{:2} - 2\frac{x^T a_{:2}}{v^T v} v = \frac{1}{\sqrt{5}} \begin{bmatrix} -4 \\ -3 \end{bmatrix},$$

and with the orthogonal matrix Q obtained from equation (5.13) as $Q = Q_1 = F^{-1}$, we have

$$Q = \frac{1}{\sqrt{5}} \begin{bmatrix} 2 & -1 \\ -1 & -2 \end{bmatrix}, \quad R = \frac{1}{\sqrt{5}} \begin{bmatrix} 5 & -4 \\ 0 & -3 \end{bmatrix}.$$

Hessenberg QR factorization

A *Hessenberg matrix* is almost triangular, in the sense that all components that are below the first subdiagonal are zero. We can use the structure of a Hessenberg matrix to design a much more efficient QR factorization algorithm using Givens rotation matrices

$$G = \begin{bmatrix} c & -s \\ s & c \end{bmatrix},$$

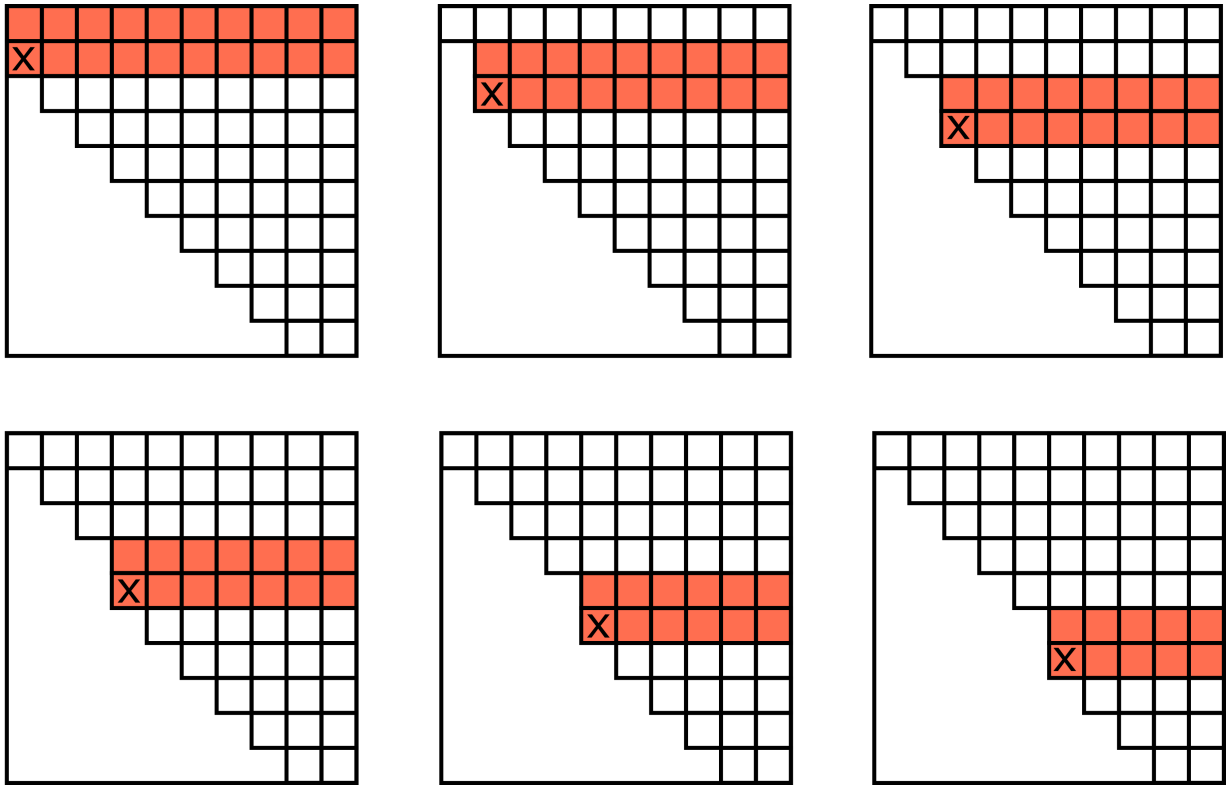
where for an arbitrary vector $x = (x_1, x_2)^T \in R^2$,

$$c = \cos(\theta) = x_1/\|x\|, \quad s = \sin(\theta) = x_2/\|x\|,$$

with the angle θ chosen such that

$$Gx = (\|x\|, 0)^T.$$

Hessenberg QR factorization

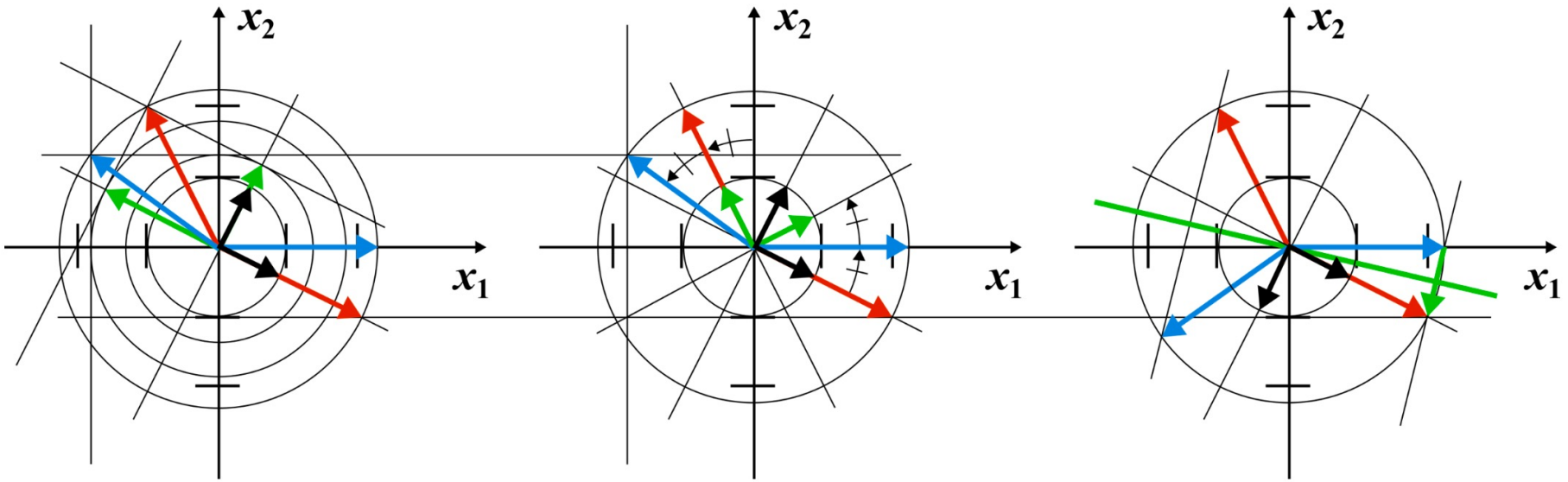


Hessenberg QR factorization

ALGORITHM 5.5. (H, G_0,..,G_n-1) = hessenberg_qr_factorization(H).
Input: Hessenberg matrix **H**.
Output: upper triangular matrix **H** and the Givens matrices **G_0**,...,**G_n-2**.

```
1: for k=0:n-2 do
2:     G_k = givens_rotation(H[k:k+1,k:k+1])
3: end for
4: return H, G_0,..., G_n-2
```

Gram-Schmidt, Hessenberg, Householder



LU factorization by Gaussian elimination

$$L_{n-1} \cdots L_2 L_1 P A = U,$$

from which we get the factorization $PA = LU$, with

$$L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1},$$

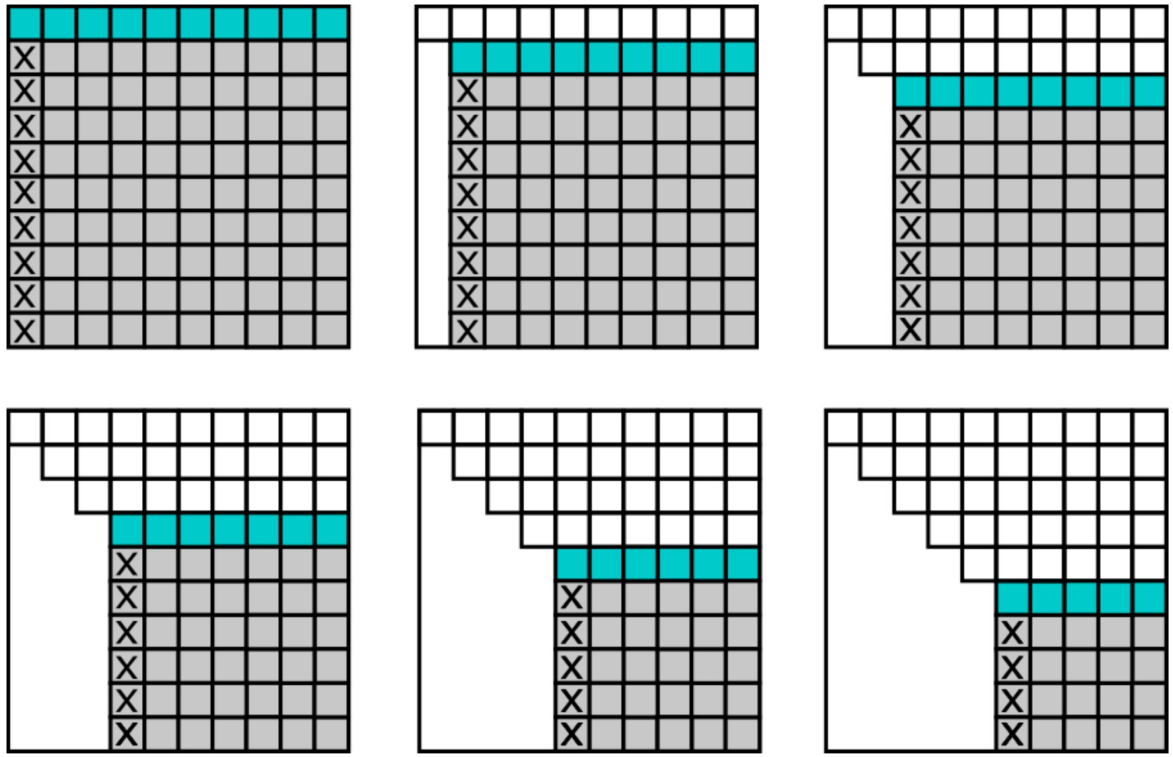
where P is a permutation matrix that represents the reordering of the rows, or *pivoting*.

LU factorization by Gaussian elimination

ALGORITHM 5.6. (L, U, P) = gaussian_elimination_with_pivoting(A).
Input: full rank nxn matrix **A**.
Output: lower triangular matrix **L**, upper triangular matrix **U**, permutation matrix **P**.

```
1:  U = A, L = I, P = I
2:  for k=0:n-1 do
3:      Find i≥k which maximizes |U[i,k]|.
4:      Interchange the rows k and i in the matrices U, L, P.
5:      for j=k+1:n-1 do
6:          L[j,k] = U[j,k]/U[k,k]
7:          U[j,k:n-1] = U[j,k:n-1] - L[j,k]*U[k,k:n-1]
8:      end for
9:  end for
10: return  L, U, P
```

LU factorization by Gaussian elimination



LU factorization by Gaussian elimination

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \qquad U = A, \quad L = P = I$$

$$u_{11} = 2$$
$$l_{21} = u_{21}/u_{11} = -1/2$$

$$u_{21} = u_{21} - l_{21}u_{11} = -1 - (-1/2)2 = 0,$$
$$u_{22} = u_{22} - l_{21}u_{12} = 2 - (-1/2)(-1) = 3/2$$

$$U = \begin{bmatrix} 2 & -1 \\ 0 & 3/2 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 \\ -1/2 & 1 \end{bmatrix}, \quad P = I$$

If A is SPD: Cholesky factorization $A = LL^T$

$$\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{2n} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} l_{11}^2 & l_{21}l_{11} & \cdots & l_{n1}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & \cdots & l_{n1}l_{21} + l_{n2}l_{22} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1}l_{11} & l_{n1}l_{21} + l_{n2}l_{22} & \cdots & l_{n1}^2 + \ldots + l_{nn}^2 \end{bmatrix}$$

If A is SPD: Cholesky factorization $A = LL^T$

$$l_{ii} = \pm\sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2},$$

$$l_{ij} = \frac{1}{l_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}l_{jk}\right), \quad i > j$$

If A is SPD: Cholesky factorization $A = LL^T$

Example 5.4. Cholesky factorization can be used for the 2×2 symmetric positive definite matrix A defined in equation (5.9). Equations (5.14)-(5.15) then give that

$$l_{11} = \sqrt{2}, \quad l_{21} = -\frac{1}{\sqrt{2}}, \quad l_{22} = \sqrt{2 - \left(-\frac{1}{\sqrt{2}}\right)^2} = \sqrt{\frac{3}{2}},$$

by choosing the positive sign in equation (5.14). The computed matrix L can be verified to be the Cholesky factorization of A, since

$$L = \begin{bmatrix} \sqrt{2} & 0 \\ -1/\sqrt{2} & \sqrt{3/2} \end{bmatrix} \quad \Rightarrow \quad LL^T = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}.$$

Matrix-vector product

ALGORITHM 3.2. b = matrix_vector_product(A, x).
Input: an nxn matrix **A** and an n vector **x**.
Output: the matrix-vector product n vector **b** = Ax.

```
1: b = 0
2: for i=0:n-1 do
3:     for j=0:n-1 do
4:         b[i] = b[i]+A[i,j]*x[j]
5:     end for
6: end for
7: return  b
```

Sparse matrix-vector product

We say that a matrix $A \in R^{n \times n}$ is sparse if most of its components are zero. If a matrix is not sparse it is dense. Whereas for a dense matrix the number of nonzero components is $\mathcal{O}(n^2)$, for a sparse matrix of full rank it is only $\mathcal{O}(n)$, which has obvious implications for the memory footprint and the efficiency of algorithms that can exploit the sparsity of a matrix. The specific

Specific data structures have been developed for sparse matrices. One such data structure is the *compressed row storage* (CRS) format, which provides an efficient representation of a sparse $m \times n$ matrix with nnz number of nonzero components. It takes the form of three arrays containing the nonzero values (val), their respective column indices (col_idx), and an array that points to the start of each row and ends with $nnz + 1$ (row_ptr).

Sparse matrix-vector product

$$A = \begin{bmatrix} 3 & 2 & 0 & 2 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 3 \end{bmatrix}$$

$$\begin{aligned} val &= [3 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 3 \quad 2 \quad 1 \quad 2 \quad 3] \\ col\_idx &= [1 \quad 2 \quad 4 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4 \quad 5 \quad 5 \quad 6] \\ row\_ptr &= [1 \quad 4 \quad 6 \quad 7 \quad 9 \quad 10 \quad 12], \end{aligned}$$

Sparse matrix-vector product

ALGORITHM 5.9. b = sparse_matrix_vector_product(A, x).
Input: a sparse mxn matrix **A**, and an n vector **x**.
Output: the matrix-vector product **b** = Ax

```
1: for i=0:n-1 do
2:     b[i]=0
3:     for j=A.row_ptr[i]:A.row_ptr[i+1]-1 do
4:         b[i]= b[i] + A.val[j]*x[A.col_idx[j]]
5:     end for
6: end for
7: return b
```

Computational intensity

The complexity of an algorithm is only one part of the computational cost. Often the cost to access slow memory through read or write operations is the dominating cost. We define the computational intensity q of an algorithm as the ratio between the average number of floating point operations f and memory references m,

$$q = f/m.$$

Hence, the total time for an algorithm to complete can be estimated as the sum

$$f\,t_f + m\,t_m = f\,t_f \left(1 + \frac{t_m}{t_f}\,\frac{1}{q}\right),$$

where t_f is the time for one floating point operation and t_m the time for one memory reference, with $t_f << t_m$. Whereas t_m/t_f is a fixed characteristic of the hardware, the total time can be reduced by increasing the computational intensity of an algorithm.

Matrix-vector product

Algorithm 3.2 computes the matrix-vector multiplication $b = Ax$, with

$$f \sim n^2$$

number of floating point operations. We read the vector x and the matrix A, and write the vector b, that is, we have

$$m = 2n + n^2$$

memory references. Therefore, the computational intensity is $q \sim n^2/n^2 = 1$. For the matrix-matrix multiplication

$$C = AB$$

Matrix-vector product

ALGORITHM 5.7. C = matrix_matrix_product(A, B).
Input: an mxp matrix **A** and an pxn matrix **B**.
Output: the matrix-matrix product mxn matrix **C** = AB.

```
1: C = 0
2: for i=0:m-1 do
3:     for j=0:n-1 do
4:         for k=0:p-1 do
5:             C[i,j] = C[i,j]+A[i,k]*B[k,j]
6:         end for
7:     end for
8: end for
9: return C
```

Matrix-matrix product

Algorithm 5.7 is identical to matrix-vector multiplication for each column vector in C. The algorithm has a complexity of $\mathcal{O}(n^3)$ and operates on data of size $3n^2$, so that ideally

$$q \sim n^3/n^2 \sim n.$$

That is, the computational intensity is $q \sim n$, compared to $q \sim 1$ for matrix-vector multiplication. This means that matrix-matrix multiplication should be efficient compared to matrix-vector multiplication. But when we analyze Algorithm 5.7 we find that the number of memory references is $2n^2 + n^3$, since each column of the matrix B is read n times, see Figure 5.6. Hence, the computational intensity is only $q \sim n^3/n^3 \sim 1$.

Matrix-matrix product

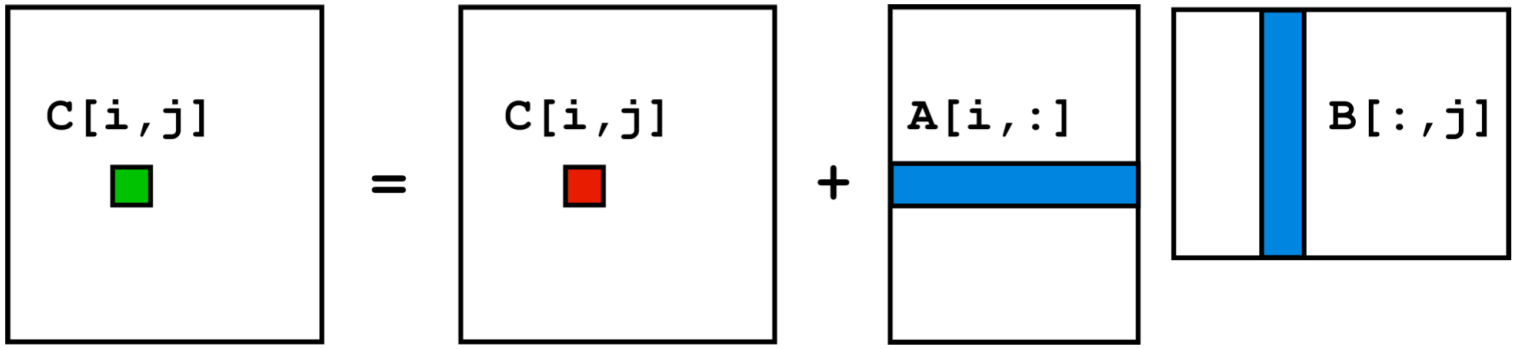


Figure 5.6. *Illustration of the matrix-matrix multiplication Algorithm 5.7, where the row A[i,:] is read once and used for the row C[i,:], whereas the columns B[:,j] are re-read for each column of C.*

Blocked matrix-matrix product

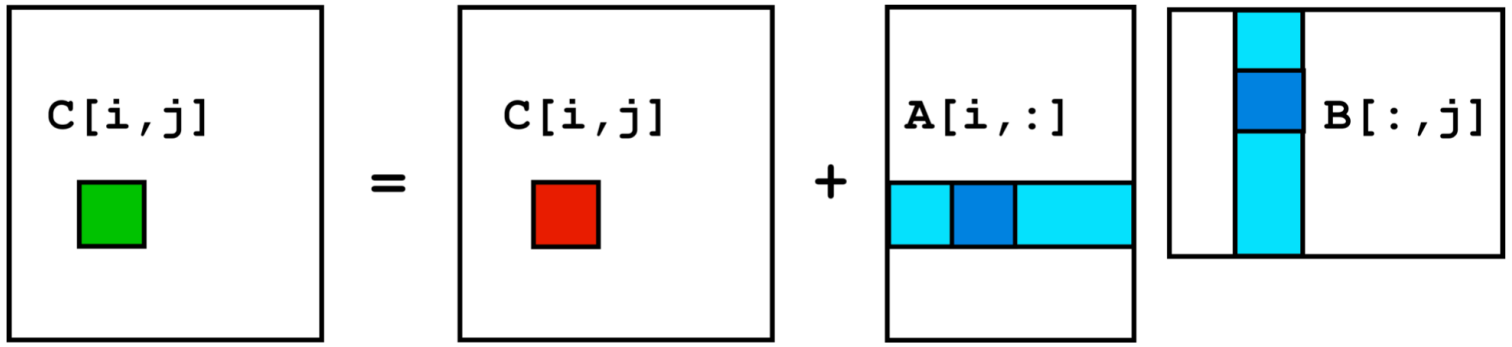


Figure 5.7. *Illustration of the blocked matrix-matrix multiplication Algorithm 5.8, where the blocks A[i,k] and B[k,j] are read once for each block C[i,j].*

Blocked matrix-matrix product

To increase the computational intensity we need to reduce the number of memory references. We can achieve this by partitioning the matrices into blocks with dimension $b = n/N$. where N is the number of blocks and n the dimension of each matrix.

The total number of memory references for matrices A, B and C are then

$$m = Nn^2 + Nn^2 + n^2 = n^2(2N + 1),$$

with gives a computational intensity of

$$q \sim n^3/n^2N \sim n/N \sim b.$$

Blocked matrix-matrix product

ALGORITHM 5.8. C = blocked_matrix_matrix_product(A, B).
Input: an mxp matrix **A** and an pxn matrix **B**.
Output: the matrix-matrix product mxn matrix **C** = AB.

```
1:  bm = m/M
2:  bn = n/N
3:  bp = p/P
    {Set block sizes.}
4:  C = 0
5:  for i=0:M-1 do
6:      for j=0:N-1 do
7:          for k=0:P-1 do
8:              C[i,j] = C[i,j] + matrix_matrix_product(A[i,k], B[k,j])
                {Block operation without need for slow memory references.}
9:          end for
10:     end for
11: end for
12: return C
```