Systems of linear equations

Introduction

A *system of linear equations* (or *linear system*) is a collection of linear equations involving the same set of unknown variables. A general system of n linear equations with m unknowns can be written as

$$\left\{\begin{array}{ccccccccc} A_{11}x_1 & + & A_{12}x_2 & + & \cdots & + & A_{1m}x_m & = & b_1 \\ A_{21}x_1 & + & A_{22}x_2 & + & \cdots & + & A_{2m}x_m & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ A_{n1}x_1 & + & A_{n2}x_2 & + & \cdots & + & A_{nm}x_m & = & b_n \end{array}\right. , \tag{1}$$

where $x_1, x_2, \ldots, x_m$ are the unknown variables, $A_{11}, A_{12}, \ldots, A_{nm}$ are the (constant) coefficients of the system, and $b_1, b_2, \ldots, b_n$ are the (constant) right-hand side terms.

The system can be equivalently written in the matrix form,

$$\mathbf{A}\mathbf{x} = \mathbf{b} , \tag{2}$$

where $\mathbf{A} \doteq \{A_{ij}\}$ is the $n \times m$ matrix of the coefficients, $\mathbf{x} \doteq \{x_j\}$ is the size-m column-vector of the unknown variables, and $\mathbf{b} \doteq \{b_i\}$ is the size-n column-vector of right-hand side terms.

A solution to a linear system is a set of values for the variables $\mathbf{x}$ which satisfies all equations.

Systems of linear equations occur quite regularly in applied mathematics. Therefore computational algorithms for finding solutions of linear systems are an important part of numerical methods.

A system of non-linear equations can often be approximated by a linear system – a helpful technique (called *linearization*) in creating a mathematical model of an otherwise a more complex system.

If $m = n$, the matrix A is called *square*. A square system has a unique solution if A is invertible.

Triangular systems

An efficient algorithm to solve a square system of linear equations numerically is to transform the original system into an equivalent *triangular system*,

$$\mathbf{T}\mathbf{y} = \mathbf{c} , \tag{3}$$

where $\mathbf{T}$ is a *triangular matrix* – a special kind of square matrix where the matrix elements either below (lower triangular) or above (upper triangular) the main diagonal are zero.

Indeed, an upper triangular system $\mathbf{U}\mathbf{y} = \mathbf{c}$ can be easily solved by *back-substitution*,

$$y_i = \frac{1}{U_{ii}} \left(c_i - \sum_{k=i+1}^{n} U_{ik} y_k \right) , \; i = n, n-1, \ldots, 1 , \tag{4}$$

where one first computes $y_n = b_n / U_{nn}$, then substitutes *back* into the previous equation to solve for y_{n-1}, and repeats through y_1.

Here is a C-function implementing in-place[1] back-substitution[2]:

```
#include <matrix.h>
#include <assert.h>
void backsub(const matrix* U, vector* b){
  int n=b->size; assert(U->size1 == U->size2 && U->size1 == n);
  for(int i = n-1; i>=0; i--){
    double sum=vector_get(b,i);
    for(int k=i+1;k<n;k++) sum -= matrix_get(U,i,k)*vector_get(b,k);
    vector_set(b,i,sum/matrix_get(U,i,i)); } }
```

For a lower triangular system $\mathbf{L}\mathbf{y} = \mathbf{c}$ the equivalent procedure is called *forward-substitution*,

$$y_i = \frac{1}{L_{ii}} \left(c_i - \sum_{k=1}^{i-1} U_{ik} y_k \right) , \; i = 1, 2, \ldots, n . \tag{5}$$

[1] here *in-place* means the right-hand side $\mathbf{c}$ is replaced by the solution $\mathbf{y}$.

[2] the functions `vector_get`, `vector_set`, and `matrix_get` are assumed to implement fetching and setting the vector and matrix elements.

Reduction to triangular form

Popular algorithms for reducing a square system of linear equations to a triangular form are *LU-decomposition* and *QR-decomposition*.

QR-decomposition

QR-decomposition is a factorization of a matrix into a product of an orthogonal matrix $\mathbf{Q}$, such that $\mathbf{Q}^\intercal\mathbf{Q} = 1$, where $^\intercal$ denotes transposition, and a right triangular matrix $\mathbf{R}$,

$$\mathbf{A} = \mathbf{QR} \,. \tag{6}$$

QR-decomposition can be used to convert a linear system $\mathbf{Ax} = \mathbf{b}$ into the triangular form (by multiplying with $\mathbf{Q}^\intercal$ from the left),

$$\mathbf{Rx} = \mathbf{Q}^\intercal\mathbf{b} \,, \tag{7}$$

which can be solved directly by back-substitution.

QR-decomposition can also be performed on non-square matrices with few long columns. Generally speaking a rectangular $n \times m$ matrix $\mathbf{A}$ can be represented as a product, $\mathbf{A} = \mathbf{QR}$, of an orthogonal $n \times m$ matrix $\mathbf{Q}$, $\mathbf{Q}^\intercal\mathbf{Q} = 1$, and a right-triangular $m \times m$ matrix $\mathbf{R}$.

QR-decomposition of a matrix can be computed using several methods, such as Gram-Schmidt orthogonalization, Householder transformation [**?**], or Givens rotation [**?**].

Gram-Schmidt orthogonalization *Gram-Schmidt orthogonalization* is an algorithm for orthogonalization of a set of vectors in a given inner product space. It takes a linearly independent set of vectors $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ and generates an orthogonal set $\mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_m\}$ which spans the same subspace as $\mathbf{A}$. The algorithm is given as

```
for i = 1 to m :
  q_i ← a_i/‖a_i‖
  for j = i + 1 to m :  a_j ← a_j − ⟨q_i|a_j⟩ q_i
```

where $\langle\mathbf{a}|\mathbf{b}\rangle$ is the inner product of two vectors, and $\|\mathbf{a}\| \doteq \sqrt{\langle\mathbf{a}|\mathbf{a}\rangle}$ is the vector's norm. This variant of the algorithm, where all remaining vectors $\mathbf{a}_j$ are made orthogonal to $\mathbf{q}_i$ as soon as the latter is calculated, is considered to be numerically stable and is referred to as *stabilized* or *modified*.

Stabilized Gram-Schmidt orthogonalization can be used to compute QR-decomposition of a matrix $\mathbf{A}$ by orthogonalization of its column-vectors $\mathbf{a}_i$ with the inner product

$$\langle\mathbf{a}|\mathbf{b}\rangle = \mathbf{a}^\intercal\mathbf{b} \equiv \sum_{k=1}^{n} (\mathbf{a})_k (\mathbf{b})_k \,, \tag{8}$$

where n is the length of column-vectors $\mathbf{a}$ and $\mathbf{b}$, and $(\mathbf{a})_k$ is the kth element of the column-vector,

```
for i = 1 to m :
  R_ii = √(a_i^⊺ a_i) ;  q_i = a_i/R_ii
  for j = i + 1 to m :
    R_ij = q_i^⊺ a_j ;  a_j = a_j − q_i R_ij .
```

After orthogonalization the matrices $\mathbf{Q} = \{\mathbf{q}_1 \ldots \mathbf{q}_m\}$ and $\mathbf{R}$ are the sought orthogonal and right-triangular factors of matrix $\mathbf{A}$.

The factorization is unique under requirement that the diagonal elements of R are positive. For a $n \times m$ matrix the complexity of the algorithm is $O(m^2 n)$.

Table (1) shows a C implementation of the stabilized Gram-Schmidt orthogonalization. The corresponding function to solve the equation

$$\mathbf{QRx} = \mathbf{b} \tag{9}$$

are shown in Table (2).

Table 1: QR-decomposition in C using stabilized Gram-Schmidt orthogonalization.

```
#include <math.h>
#include <matrix.h>
void qrdec(matrix* A, matrix* R) { // A is replaced with Q
  for(int i=0; i<A->size2; i++) {
    vector* ai = matrix_get_column(A,i);
    double r = vector_dot( ai, ai );
    matrix_set(R,i,i,sqrt(r));
    vector_scale(ai,1/sqrt(r));
    for(int j=i+1; j<A->size2; j++) {
      vector* aj = matrix_get_column(A,j);
      double s = vector_dot( ai, aj );
      vector_add( aj, -s, ai );
      matrix_set(R,i,j,s); } } }
```

Table 2: C-functions to perform QR-backsubstitution of equation (9)

```
#include <matrix.h>
#include <assert.h>
void backsub(const matrix* U, vector* b);

vector* qrback(const matrix* Q, const matrix* R, const vector* b)
{ vector* x = matrixT_times_vector(Q,b);
  backsub(R,x); return x;
}
void qrback_inplace(const matrix* Q, const matrix* R, vector* b)
{ vector *x = matrixT_times_vector(Q,b);
  backsub(R,x);
  for(int i=0;i<x->size;i++)vector_set(b,i,vector_get(x,i));
}
```

Householder transformation A square matrix $\mathbf{H}$ of the form

$$\mathbf{H} = 1 - \frac{2}{\mathbf{u}^\intercal \mathbf{u}} \mathbf{u}\mathbf{u}^\intercal \tag{10}$$

is called *Householder matrix*, where the vector $\mathbf{u}$ is called a *Householder vector*. Householder matrices are symmetric and orthogonal,

$$\mathbf{H}^\intercal = \mathbf{H}\,,\ \mathbf{H}^\intercal\mathbf{H} = 1\,. \tag{11}$$

The transformation induced by the Householder matrix on a given vector a,

$$\mathbf{a} \to \mathbf{H}\mathbf{a}\,, \tag{12}$$

is called a *Householder transformation* or *Householder reflection*. The transformation changes the sign of the affected vector's component in the $\mathbf{u}$ direction, or, in other words, makes a reflection of the vector about the hyperplane perpendicular to $\mathbf{u}$, hence the name.

Householder transformation can be used to zero selected components of a given vector $\mathbf{a}$. For example, one can zero all components but the first one, such that

$$\mathbf{H}\mathbf{a} = \gamma \mathbf{e}_1\,, \tag{13}$$

where γ is a number and $\mathbf{e}_1$ is the unit vector in the first direction. The factor γ can be easily calculated,

$$\|\mathbf{a}\|^2 \doteq \mathbf{a}^\intercal\mathbf{a} = \mathbf{a}^\intercal\mathbf{H}^\intercal\mathbf{H}\mathbf{a} = (\gamma\mathbf{e}_1)^\intercal(\gamma\mathbf{e}_1) = \gamma^2\,, \tag{14}$$

$$\Rightarrow\ \gamma = \pm\|a\|\,. \tag{15}$$

To find the Householder vector, we notice that

$$\mathbf{a} = \mathbf{H}^\intercal\mathbf{H}\mathbf{a} = \mathbf{H}^\intercal\gamma\mathbf{e}_1 = \gamma\mathbf{e}_1 - \frac{2(\mathbf{u})_1}{\mathbf{u}^\intercal\mathbf{u}}\mathbf{u}\,, \tag{16}$$

$$\Rightarrow \frac{2(\mathbf{u})_1}{\mathbf{u}^\intercal\mathbf{u}}\mathbf{u} = \gamma\mathbf{e}^1 - \mathbf{a}\,, \tag{17}$$

where $(\mathbf{u})_1$ is the first component of the vector $\mathbf{u}$. One usually chooses $(\mathbf{u})_1 = 1$ (for the sake of the possibility to store the other components of the Householder vector in the zeroed elements of the vector $\mathbf{a}$) and stores the factor

$$\frac{2}{\mathbf{u}^\intercal\mathbf{u}} \equiv \tau \tag{18}$$

separately. With this convention one readily finds τ from the first component of equation (17),

$$\tau = \gamma - (\mathbf{a})_1\,. \tag{19}$$

where $(\mathbf{a})_1$ is the first element of the vector $\mathbf{a}$. For the sake of numerical stability the sign of γ has to be chosen opposite to the sign of $(\mathbf{a})_1$,

$$\gamma = -\mathrm{sign}\left((\mathbf{a})_1\right)\|\mathbf{a}\|\,. \tag{20}$$

Finally, the Householder reflection, which zeroes all component of a vector $\mathbf{a}$ but the first, is given as

$$\mathbf{H} = 1 - \tau\mathbf{u}\mathbf{u}^\intercal\,,\ \tau = -\mathrm{sign}((\mathbf{a})_1)\|\mathbf{a}\| - (\mathbf{a})_1\,,\ (\mathbf{u})_1 = 1\,,\ (\mathbf{u})_{i>1} = -\frac{1}{\tau}(\mathbf{a})_i\,. \tag{21}$$

Now, a QR-decomposition of an $n \times n$ matrix $\mathbf{A}$ by Householder transformations can be performed in the following way:

1. Build the size-n Householder vector $\mathbf{u}_1$ which zeroes the sub-diagonal elements of the first column of matrix $\mathbf{A}$, such that

$$\mathbf{H}_1\mathbf{A} = \left[\begin{array}{c|ccc} \star & \star & \dots & \star \\ \hline 0 & & & \\ \vdots & & \mathbf{A}_1 & \\ 0 & & & \end{array}\right]\,, \tag{22}$$

   where $\mathbf{H}_1 = 1 - \tau_1\mathbf{u}_1\mathbf{u}_1^\intercal$ and where $\star$ denotes (generally) non-zero matrix elements. In practice one does not build the matrix $\mathbf{H}_1$ explicitly, but rather calculates the matrix $\mathbf{H}_1\mathbf{A}$ in-place, consecutively applying the Householder reflection to columns the matrix $\mathbf{A}$, thus avoiding computationally expensive matrix-matrix operations. The zeroed sub-diagonal elements of the first column of the matrix $\mathbf{A}$ can be used to store the elements of the Householder vector $\mathbf{u}_1$ while the factor τ_1 has to be stored separately in a special array. This is the storage scheme used by LAPACK and GSL.

2. Similarly, build the size-$(n-1)$ Householder vector $\mathbf{u}_2$ which zeroes the sub-diagonal elements of the first column of matrix $\mathbf{A}_1$ from eq. (22). With the transformation matrix $\mathbf{H}_2$ defined as

$$\mathbf{H}_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & 1 - \tau_2\mathbf{u}_2\mathbf{u}_2^\intercal & \\ 0 & & & \end{array}\right]\,. \tag{23}$$

   the two transformations together zero the sub-diagonal elements of the two first columns of matrix $\mathbf{A}$,

$$\mathbf{H}_2\mathbf{H}_1\mathbf{A} = \left[\begin{array}{cc|ccc} \star & \star & \star & \cdots & \star \\ 0 & \star & \star & \cdots & \star \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & \mathbf{A}_3 & \\ 0 & 0 & & & \end{array}\right]\,, \tag{24}$$

3. Repeating the process zero the sub-diagonal elements of the remaining columns. For column k the corresponding Householder matrix is

$$\mathbf{H}_k = \left[\begin{array}{c|c} \mathbf{I}_{k-1} & 0 \\ \hline 0 & 1 - \tau_k\mathbf{u}_k\mathbf{u}_k^\intercal \end{array}\right]\,, \tag{25}$$

where $\mathbf{I}_{k-1}$ is a unity matrix of size $k-1$, $\mathbf{u}_k$ is the size-(n-k+1) Householder vector that zeroes the sub-diagonal elements of matrix $\mathbf{A}_{k-1}$ from the previous step. The corresponding transformation step is

$$\mathbf{H}_k \dots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} = \left[\begin{array}{c|c} \mathbf{R}_k & \star \\ \hline 0 & \mathbf{A}_k \end{array}\right] , \tag{26}$$

where $\mathbf{R}_k$ is a size-k right-triangular matrix.

After $n-1$ steps the matrix $\mathbf{A}$ will be transformed into a right triangular matrix,

$$\mathbf{H}_{n-1} \cdots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} = \mathbf{R} . \tag{27}$$

4. Finally, introducing an orthogonal matrix $\mathbf{Q} = \mathbf{H}_1^\intercal \mathbf{H}_2^\intercal \dots \mathbf{H}_{n-1}^\intercal$ and multiplying eq. (27) by $\mathbf{Q}$ from the left, we get the sought QR-decomposition,

$$\mathbf{A} = \mathbf{Q}\mathbf{R} . \tag{28}$$

In practice one does not explicitly builds the $\mathbf{Q}$ matrix but rather applies the successive Householder reflections stored during the decomposition.

Givens rotations A Givens rotation is a transformation in the form

$$\mathbf{A} \to \mathbf{G}(p, q, \theta)\mathbf{A} , \tag{29}$$

where $\mathbf{A}$ is the transformed object—matrix of vector—and $\mathbf{G}(p, q, \theta)$ is the Givens rotation matrix (also known as Jacobi rotation matrix): an orthogonal matrix in the form

$$G(p,q,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{array}{l} \\ \\ \leftarrow \text{row } p \\ \\ \leftarrow \text{row } q \\ \\ \end{array} . \tag{30}$$

When a Givens rotation matrix $\mathbf{G}(p, q, \theta)$ multiplies a vector, only elements with indices p and q are affected. Restricting our attention to these two elements, say a and b, the Givens rotation is given explicitely as

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a\cos\theta + b\sin\theta \\ -a\sin\theta + b\cos\theta \end{bmatrix} . \tag{31}$$

Apparently, the rotation can zero the lowest element, if the angle θ is chosen as

$$\tan\theta = \frac{b}{a} \quad \Rightarrow \quad \theta = \operatorname{atan2}(b, a) . \tag{32}$$

A sequence of Givens rotations can zero all elements of a matrix below the main diagonal. Which obviously amounts to QR-decomposition of the matrix with $\mathbf{Q}$ being the product of all the applied Givens matrices.

Since each Givens rotation only affects two rows of the matrix it is possible to apply a set of rotations in parallel. Givens rotations are also more efficient on sparse matrices.

LU-decomposition

LU-decomposition is a factorization of a square matrix $\mathbf{A}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$,

$$\mathbf{A} = \mathbf{L}\mathbf{U} . \tag{33}$$

The linear system $\mathbf{Ax} = \mathbf{b}$ after LU-decomposition of the matrix $\mathbf{A}$ becomes $\mathbf{LUx} = \mathbf{b}$ and can be solved by first solving $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ and then $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$ with two runs of forward and backward substitutions.

If $\mathbf{A}$ is an $n \times n$ matrix, the condition (33) is a set of n^2 equations,

$$\sum_{k=1}^{n} L_{ik}U_{kj} = A_{ij} \Big|_{i,j=1\ldots n} , \tag{34}$$

for $n^2 + n$ unknown elements of the triangular matrices $\mathbf{L}$ and $\mathbf{U}$. The decomposition is thus not unique.

Usually the decomposition is made unique by providing extra n conditions e.g. by the requirement that the elements of the main diagonal of the matrix $\mathbf{L}$ are equal one,

$$L_{ii} = 1 \, , \; i = 1 \ldots n \, . \tag{35}$$

The system (34) with the extra conditions (35) can then be easily solved row after row using the *Doolittle's algorithm*,

```
for i = 1...n :
   L_ii = 1
   for j = i...n :   U_ij = A_ij − Σ_{k<i} L_ik U_kj
   for j = i+1...n :   L_ji = (1/U_ii) ( A_ji − Σ_{k<j} L_jk U_ki )
```

In a slightly different *Crout's algorithm* it is the matrix $\mathbf{U}$ that has unit diagonal elements,

```
for i = 1...n :
   U_ii = 1
   for j = i...n :   L_ji = A_ji − Σ_{k<i} L_jk U_ki
   for j = i+1...n :   U_ij = (1/L_ii) ( A_ji − Σ_{k<j} L_jk U_ki )
```

Without a proper ordering (permutations) in the matrix, the factorization may fail. For example, it is easy to verify that $A_{11} = L_{11}U_{11}$. If $A_{11} = 0$, then at least one of L_{11} and U_{11} has to be zero, which implies either $\mathbf{L}$ or $\mathbf{U}$ is singular, which is impossible if $\mathbf{A}$ is non-singular. This is however only a procedural problem. It can be removed by simply reordering the rows of $\mathbf{A}$ so that the first element of the permuted matrix is nonzero (or, even better, the largest in absolute value among all elements of the column below the diagonal). The same problem in subsequent factorization steps can be removed in a similar way. Such algorithm is refered to as *partial pivoting*. It requires an extra integer array to keep track of row permutations.

Determinant of a matrix

LU- and QR-decompositions allow $O(n^3)$ calculation of the determinant of a square matrix. Indeed, for the LU-decomposition,

$$\det \mathbf{A} = \det \mathbf{LU} = \det \mathbf{L} \det \mathbf{U} = \det \mathbf{U} = \prod_{i=1}^{n} U_{ii} \, . \tag{36}$$

For the QR-decomposition

$$\det \mathbf{A} = \det \mathbf{QR} = \det \mathbf{Q} \det \mathbf{R} \, . \tag{37}$$

Since $\mathbf{Q}$ is an orthogonal matrix $(\det \mathbf{Q})^2 = 1$ and therefore

$$|\det \mathbf{A}| = |\det \mathbf{R}| = \left| \prod_{i=1}^{n} R_{ii} \right| \, . \tag{38}$$

Matrix inverse

The inverse $\mathbf{A}^{-1}$ of a square $n \times n$ matrix $\mathbf{A}$ can be calculated by solving n linear equations

$$\mathbf{A}\mathbf{x}_i = \mathbf{e}_i \Big|_{i=1\ldots n} , \tag{39}$$

where $\mathbf{e}_i$ is a column where all elements are equal zero except for the element number i, which is equal one; that is, columns $\mathbf{e}_i$ form a unity matrix. The matrix made of columns $\mathbf{x}_i$ is apparently the inverse of $\mathbf{A}$.

A C implementation of this algorithm using QR-decomposition is shown in Table (3).

Table 3: Matrix inverse with QR-decomposition in C

```
#include <matrix.h>
#include <assert.h>
void qrdec(matrix*A, matrix*R);
void qrback_inplace(const matrix*Q, const matrix*R, vector*b);
matrix* qrinverse(const matrix* A)
{ int n=A->size1; assert(n==A->size2);
  matrix *R = matrix_alloc(n,n), *Q=matrix_copy(A); qrdec(Q,R);
  matrix *I = unit_matrix(n);
  for(int i=0;i<n;i++) qrback_inplace(Q,R,matrix_get_column(I,i));
  return I;
}
```