

Lecture 6: Using a QR Decomposition to Solve Linear Systems

1.3 Solving linear systems of equations using the QR decomposition.

Suppose $\mathbf{A} \in \mathbb{C}^{n \times n}$ is a nonsingular (i.e., invertible) matrix, and we wish to solve the system of linear equations $\mathbf{Ax} = \mathbf{b}$ for the unknown $\mathbf{x} \in \mathbb{C}^n$. The QR decomposition, $\mathbf{A} = \mathbf{QR}$, allows us to transform this general system into a simpler problem:

$$\mathbf{Ax} = \mathbf{b} \quad \iff \quad \mathbf{QRx} = \mathbf{b} \quad \iff \quad \mathbf{Rx} = \mathbf{Q}^*\mathbf{b}.$$

This final system involves the upper triangular matrix \mathbf{R} , and thus can be quickly solved with *backward substitution*.

1.3.1 Solving upper triangular linear systems.

Now we address the simple problem of solving a linear system of the form $\mathbf{Rx} = \mathbf{c}$, where $\mathbf{R} \in \mathbb{C}^{n \times n}$ is upper triangular, i.e., $r_{jk} = 0$ if $j > k$. (Of course, \mathbf{c} here would be replaced by $\mathbf{Q}^*\mathbf{b}$ in the context of the original linear system $\mathbf{Ax} = \mathbf{b}$.) This equation takes the form

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}. \quad (1)$$

The last row of this vector equation is scalar equation

$$r_{nn}x_n = c_n,$$

from which we can immediately find x_n :

$$x_n = c_n/r_{nn}.$$

The penultimate row of (1) gives

$$r_{n-1,n-1}x_{n-1} + r_{n-1,n}x_n = c_{n-1}.$$

Since we have already computed x_n , the only remaining unknown is x_{n-1} , for which we find

$$x_{n-1} = \frac{1}{r_{n-1,n-1}}(c_{n-1} - r_{n-1,n}x_n).$$

We find x_{n-2}, \dots, x_1 similarly:

$$x_j = \frac{1}{r_{jj}} \left(c_j - \sum_{k=j+1}^n r_{j,k}x_k \right).$$

Since we work from the bottom of (1) up, this procedure is called *back substitution*. Clearly it is much simpler than applying Gaussian elimination to a general (non-triangular) linear system $\mathbf{Ax} = \mathbf{b}$: all the hard work was accomplished when we computed the QR factorization.

The following MATLAB code gives an efficient implementation of back substitution, written in a concise form. Can you follow it?

```

function x = backsolve(R,c)
% Solves R*x = c for x, where R is a nonsingular (square) upper triangular matrix.
[n n] = size(R);
x = zeros(n,1);
for j=n:-1:1
    x(j) = (c(j) - R(j,j+1:n)*x(j+1:n))/R(j,j);
end

```

In the first pass through the `for` loop, we have `j=n`, so the range `j+1:n` is empty. Fear not; MATLAB is smart enough to handle this: `R(j,j+1:n)` and `x(j+1:n)` are 1-by-0 and 0-by-1 empty vectors, and MATLAB has the convention that the product of such empty vectors is zero.

1.3.2 Overall complexity of solving linear systems.

What is the total computational cost of solving $\mathbf{Ax} = \mathbf{b}$ using the QR decomposition, followed by back substitution?

In the last lecture we determined that roughly

$$2mn^2 - \frac{2}{3}n^3$$

floating point operations are required to compute a QR factorization with Householder reflectors. When \mathbf{A} is square, i.e., $m = n$, this simplifies to $\frac{4}{3}n^3$ operations.

To the cost of the factorization, one needs to add the expense of forming $\mathbf{c} = \mathbf{Q}^*\mathbf{x}$ and performing back substitution. Can you work out the number of floating point operations required for these operations from the MATLAB algorithms given in this lecture and the last?

In many applications, one is presented with a sequence of linear systems of the form

$$\mathbf{Ax}_k = \mathbf{b}_k, \quad k = 1, 2, \dots,$$

where \mathbf{b}_k might depend on \mathbf{x}_{k-1} , the solution to the previous system. Once we have computed a QR factorization for \mathbf{A} , how expensive is it to solve p linear systems of the form $\mathbf{Ax}_k = \mathbf{b}_k$?

Many of you know that one can solve linear systems in MATLAB with the *backslash command*: `x=A\b` solves $\mathbf{Ax} = \mathbf{b}$. Before solving this system, MATLAB automatically checks if the matrix is upper triangular or has other special structure to exploit. (You can improve performance by explicitly telling MATLAB that your coefficient matrix has special structure with the `linsolve` command.) The following codes each solve 100 systems of the form $\mathbf{Ax}_k = \mathbf{b}_k$ with \mathbf{A} a random 500×500 matrix. Do you see why the code on the left runs about ten times faster on my laptop?

```

n = 500; p = 100;
A = randn(n); B = randn(n,p);
tic
[Q,R] = qr(A);
for k=1:p
    b = B(:,k);
    x = R\(Q'*b);
end
fprintf('elapsed time = %10.7f\n', toc)

```

```

n = 500; p = 100;
A = randn(n); B = randn(n,p);
tic
for k=1:p
    b = B(:,k);
    x = A\b;
end
fprintf('elapsed time = %10.7f\n', toc)

```