STAT 513/413: Lecture 5
Another bit of linear algebra

(solving equations)

Recall Lecture 4: solving equations

The theory for a linear model $y \sim X\beta$ may suggest to obtain the least squares estimates via the formula $b = (X^\top X)^{-1} X^\top y$

but experts in numerical computations know that it should be done rather via solving (the system of equations) $(X^\top X)b = X^\top y$

and experts in statistical computing know that the best way is to do it via so-called QR decomposition (MATLAB “backslash” operator) applied directly to the matrix X rather than $X^\top X$


```
b <- qr.solve(X, y)
```

Also, `qr.solve` can solve systems of equations $Ax = y$ when A is not square matrix

Solving systems of linear equations: Example 1

This will be our Example 1:

$x_1 + 2x_2 = 5$ we know how to solve it: $x_1 = 3$, $x_2 = 1$

$x_1 - \ \ x_2 = 2$

A sophisticated way to write the system: $Ax = b$

where $A = \begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix}$ $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and $b = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$

Subtracting first line from the second and leaving the first one as is we get the upper triangular form of the system

$x_1 + 2x_2 = 5$ which is easy to solve: first obtain

$\quad - 3x_2 = -3$ $x_2 = 1$ from the second equation

and then substitute it into the first: $x_1 + 2 = 5$

which yields $x_1 = 3$

So, (upper) triangular systems are solved easily by *backsolving*

Systems: determined, under- and overdetermined

Undetermined system has more than one solution

$x_1 + 2x_2 = 5$	this is an *undetermined* system
$2x_1 + 4x_2 = 10$	there are not enough *independent* equations

Such a system cannot be solved: we can only deduce something like a relationship giving the form of all possible solutions: here it is $x_1 = 5 - 2x_2$. This is typically not a task suitable for doing by a computer

Every system that has less equations than unknowns is undetermined. Nonetheless, as we can see, even systems with enough equations (or more) may be undetermined

Determined systems

A *determined* system: there is a unique solution to it

Apparently, a determined system has to have at least that many equations as unknowns. Usually it has the same number of both (like our Example 1), but it is not a rule: it can happen that there are more equations than unknowns

$x_1 + 2x_2 = 5$	this is a determined system
$x_1 - \ \ x_2 = 2$	number of independent equations just right
$2x_1 - 2x_2 = 4$	this one is a multiple of the previous one

This systems has (also) a unique solution $x_1 = 3$ and $x_2 = 1$

An example with more equations than unknowns

We will do this when computing invariant probability of a Markov chain. The first attempt is

```
> P = matrix(c(0.5,0.1,0.5,0.9),2,2)
> P             # note: a transition matrix of a Markov chain
     [,1] [,2]
[1,]  0.5  0.5
[2,]  0.1  0.9
> A = diag(2) - t(P)
> solve(A,c(0,0))
Error in solve.default(A, c(0, 0)) :
  system is computationally singular: reciprocal condition number = 2
> A             # indeed
     [,1] [,2]
[1,]  0.5 -0.1
[2,] -0.5  0.1
```


Indeed, we are solving for probabilities p_1 and p_2; we should add the equation $p_1 + p_2 = 1$ (otherwise any multiple of those is a solution too). After that, the system is determined, but its matrix is not square

Example 2

```
> A=rbind(A,c(1,1))
> A
     [,1] [,2]
[1,]  0.5 -0.1
[2,] -0.5  0.1
[3,]  1.0  1.0
> solve(A,c(0,0,1))
Error in solve.default(A, c(0, 0, 1)) : 'a' (3 x 2) must be square
> qr.solve(A,c(0,0,1))
[1] 0.1666667 0.8333333
```


This will be our Example 2

And now: what is this `qr.solve`?

Overdetermined systems

Overdetermined system has no solution

$$\begin{aligned} x_1 + 2x_2 &= 5 \\ x_1 - x_2 &= 2 \\ x_1 + 2x_2 &= 4 \end{aligned}$$

this is an *overdetermined* system

number of independent equations too big

Typically, an overdetermined system has more equations than unknowns - but it is not a rule; in fact, an overdetermined system may have any number of equations, starting from two

We can see that there are no x_1, x_2 satisfying *all* the equations: so in the classical sense, we cannot solve the system.

We may, however, look for some solution satisfying the system in the approximate sense

Approximate solutions of overdetermined systems

We are looking for x such that $Ax \doteq b$

where it is not really clear what $\doteq$ has to mean, but out of many possibilities we choose the mathematically and computationally convenient one: we look for x that makes $\|Ax - b\|_2^2$ minimal

Making $\|Ax - b\|_2^2$ is equivalent to making $\|Ax - b\|_2$ minimal, but the former

$$\|u\|_2^2 = u^\top u = u_1^2 + u_2^2 + \ldots u_p^2$$

is easier to handle than the latter

$$\|u\|_2 = \sqrt{u^\top u} = (u_1^2 + u_2^2 + \ldots u_p^2)^{1/2}$$

QR decomposition (full version)

Motivation: operations performed on the left- and right-hand side of the system $Ax = b$ can be interepreted as multiplying by certain matrices; for numerical stability, it is best if these matrices are orthogonal; product of orthogonal matrices is again an orthogonal matrix

QR-decomposition: writing A in a form $A = QR$

where Q is orthogonal, $Q^\top Q = QQ^\top = I$ and R is upper triangular

There are several ways of computing the QR decomposition, connected with the names of Gram-Schmidt, Householder, Givens - we do not cover this here

How is this used for solving equations?

We start with

$$Ax = y$$

which is actually, due to the fact that $A = QR$

$$QRx = y$$

On multiplying both sides from left by $Q^\top$ we obtain

$$Q^\top QRx = Q^\top y \qquad \text{that is,}$$

$$Rx = Q^\top y$$

and since R is now upper triangular, we can obtain x by backsolving

Economy version

If A is a square, $p \times p$ matrix, then everything is easy: both Q and R are square matrices with the same dimension $p \times p$

But A does not have to be square: every matrix has a QR decomposition, with possible singularity reflected in R. If A is $p \times q$, then Q has to be $p \times p$ (orthogonal matrix must be square, otherwise $Q^\top Q = QQ^\top$ could not happen) and R is thus $p \times q$

An important case is when $p > q$, then the last $p - q$ rows of R are zeros. In such a case, it may be of interest also to consider the "economy" ("thin", "reduced") version:

$\dot{Q}$ is $p \times q$ and $\dot{R}$ is $q \times q$; of course, then

$\dot{Q}$ cannot be orthogonal: it still holds true that $\dot{Q}^\top \dot{Q} = I_q$

but nothing for $\dot{Q}\dot{Q}^\top$

Some mathematical analysis perhaps

We can analyze the connection between the full Q and economy $\dot{Q}$ using blockwise matrix calculations. Knowing that $\dot{Q}$ is a part of Q, we can write $Q = \begin{pmatrix} \dot{Q} & \tilde{Q} \end{pmatrix}$. When we multiply Q by R, then the corresponding blocks of R are $\dot{R}$ and O, the latter block containing exclusively zeros. (That follows just by the analysis of dimensions and the fact that R is upper triangular.) We have

$$A = QR = \begin{pmatrix} \dot{Q} & \tilde{Q} \end{pmatrix} \begin{pmatrix} \dot{R} \\ O \end{pmatrix} = \dot{Q}\dot{R} + \tilde{Q}O = \dot{Q}\dot{R}$$

We have also

$$I_p = Q^\top Q = \begin{pmatrix} \dot{Q}^\top \\ \tilde{Q}^\top \end{pmatrix} \begin{pmatrix} \dot{Q} & \tilde{Q} \end{pmatrix} = \begin{pmatrix} \dot{Q}^\top \dot{Q} & \dot{Q}^\top \tilde{Q} \\ \tilde{Q}^\top \dot{Q} & \tilde{Q}^\top \tilde{Q} \end{pmatrix} = \begin{pmatrix} I_q & O \\ O & I_{p-q} \end{pmatrix}$$

which shows that $\dot{Q}^\top \dot{Q} = I_q$. Finally

$$I_P = QQ^\top = \begin{pmatrix} \dot{Q} & \tilde{Q} \end{pmatrix} \begin{pmatrix} \dot{Q}^\top \\ \tilde{Q}^\top \end{pmatrix} = \dot{Q}\dot{Q}^\top + \tilde{Q}\tilde{Q}^\top$$

but that does not say anything interesting

The default in Matlab is the full version

```
>> A=[0.5 -1; -0.5 0.1; 1 1]

A =
     0.5000   -1.0000
    -0.5000    0.1000
     1.0000    1.0000

>> [Q R]=qr(A)

Q =
    -0.4082   -0.8398    0.3578
     0.4082    0.1826    0.8944
    -0.8165    0.5112    0.2683


R =
    -1.2247   -0.3674
          0    1.3693
          0         0
```

Economy version has to be specifically asked for

```
>> [Q R]=qr(A,0)

Q =

   -0.4082   -0.8398
    0.4082    0.1826
   -0.8165    0.5112


R =

   -1.2247   -0.3674
         0    1.3693
```


In R, it is the other way round: the default is the economy version

Let us try it on square matrices first

```
> A=matrix(c(1,1,2,-1),nrow=2)
> A                       # remember? the matrix from Example 1
     [,1] [,2]
[1,]    1    2
[2,]    1   -1
> solve(A,c(5,2))    # and here Example 1 is solved
[1] 3 1
> qr(A)
$qr
           [,1]       [,2]
[1,] -1.4142136 -0.7071068
[2,]  0.7071068 -2.1213203

$rank
[1] 2
...
> qr(A)$qr
           [,1]       [,2]
[1,] -1.4142136 -0.7071068
[2,]  0.7071068 -2.121320
```

After reading the help `?qr`

```
> Q=qr.Q(qr(A))
> Q
           [,1]       [,2]
[1,] -0.7071068 -0.7071068
[2,] -0.7071068  0.7071068
> R=qr.R(qr(A))
> R
          [,1]       [,2]
[1,] -1.414214 -0.7071068
[2,]  0.000000 -2.1213203
```


And finally check it out

```
> Q %*% R
     [,1] [,2]
[1,]    1    2
[2,]    1   -1
```

Finishing Example 1

After transforming to $Q^\top A = Q^\top y$, we get

```
> t(Q) %*% A          # left-hand side t(Q) %*% c(5,2))
              [,1]       [,2]
[1,] -1.414214e+00 -0.7071068
[2,]  1.110223e-16 -2.1213203
> t(Q) %*% c(5,2)    # right-hand side
          [,3]
[1,] -4.949747
[2,] -2.121320
```


And then we do the backsolving

```
> (t(Q) %*% c(5,2))[2] / R[2,2]
[1] 1
> ((t(Q) %*% c(5,2))[1] - R[1,2]*x1)/R[1,1]
[1] 3
```

It works!

Example 2

```
> A
     [,1] [,2]
[1,]  0.5 -0.1
[2,] -0.5  0.1
[3,]  1.0  1.0
> Q=qr.Q(qr(A))
> Q
           [,1]       [,2]
[1,] -0.4082483  0.5773503
[2,]  0.4082483 -0.5773503
[3,] -0.8164966 -0.5773503
> round(t(Q) %*% Q,digits=3)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Example 2 continued

```
> round(Q %*% t(Q),digits=2)
     [,1] [,2] [,3]
[1,]  0.5 -0.5    0
[2,] -0.5  0.5    0
[3,]  0.0  0.0    1
> R=qr.R(qr(A))
> R
          [,1]       [,2]
[1,] -1.224745 -0.7348469
[2,]  0.000000 -0.6928203
> Q %*% R
     [,1] [,2]
[1,]  0.5 -0.1
[2,] -0.5  0.1
[3,]  1.0  1.0
```

Example 2 finished

```
> LHS=t(Q) %*% A
> RHS=t(Q) %*% c(0,0,1)
> cbind(LHS,RHS)
              [,1]       [,2]       [,3]
[1,] -1.224745e+00 -0.7348469 -0.8164966
[2,]  2.220446e-16 -0.6928203 -0.5773503
> x2=RHS[2]/LHS[2,2]
> x2
[1] 0.8333333
> x1=(RHS[1]-LHS[1,2]*x2)/LHS[1,1]
> x1
[1] 0.1666667
> qr.solve(A,c(0,0,1))
[1] 0.1666667 0.8333333
```


Note that both systems have been determined, but only the matrix of the first one was square. So, QR decomposition can nicely solve determined linear systems, even when the matrix is not square. However, it can do more: overdetermined systems.

Overdetermined system: Example 3

```
> A=cbind(c(1,1,1),c(1,2,3))
> A
     [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    1    3
> y=c(1,4,0)
```

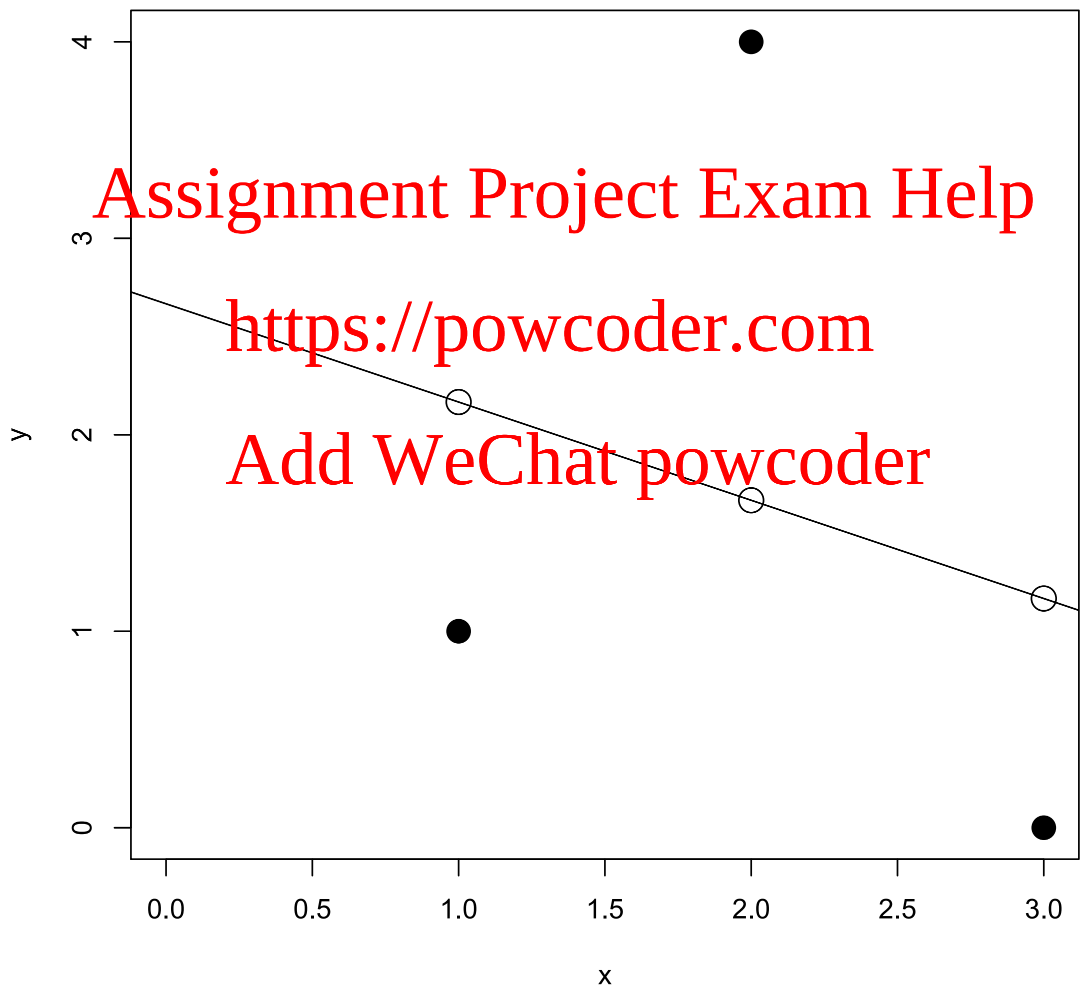

This system is clearly overdetermined - but we can solve it using QR decomposition as a determined one; but while it looks the same when running R commands, the mathematics is different here


```
> qr.solve(A,y)
[1]  2.666667 -0.500000
> abline(qr.solve(A,y))
> A %*% qr.solve(A,y)
         [,1]
[1,] 2.166667
[2,] 1.666667
[3,] 1.166667
```

So, we do not have $Ax = y$, but only $Ax \doteq y$

What goes on here?

In fact, we are solving a regression problem: fitting line through $(1, 1)$, $(2, 4)$, $(3, 0)$ via least squares (find the fitted values!)



Indeed

```
> Q=qr.Q(qr(A))
> R=qr.R(qr(A))
> Q
           [,1]          [,2]
[1,] -0.5773503  7.071068e-01
[2,] -0.5773503  2.775558e-16
[3,] -0.5773503 -7.071068e-01
> R
          [,1]      [,2]
[1,] -1.732051 -3.464102
[2,]  0.000000 -1.414214
> LHS=t(Q) %*% A
> RHS=t(Q) %*% y
> cbind(LHS,RHS)
              [,1]      [,2]       [,3]
[1,] -1.732051e+00 -3.464102 -2.8867513
[2,]  4.440892e-16 -1.414214  0.7071068
> b2=RHS[2]/LHS[2,2]
> b1=(RHS[1]-LHS[1,2]*b2)/LHS[1,1]
> c(b1,b2)
[1]  2.666667 -0.500000
```

Actually, full QR is in R too

```
> Q=qr.Q(qr(A),complete=TRUE)
> R=qr.R(qr(A),complete=TRUE)
> Q
           [,1]          [,2]       [,3]
[1,] -0.5773503  7.071068e-01  0.4082483
[2,] -0.5773503  2.775558e-16 -0.8164966
[3,] -0.5773503 -7.071068e-01  0.4082483
> R
          [,1]      [,2]
[1,] -1.732051 -3.464102
[2,]  0.000000 -1.414214
[3,]  0.000000  0.000000
> Q %*% R
     [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    1    3
```

So, why does it work for overdetermined systems?

Some first heuristics: check, for some arbitrary x (perhaps for several ones if you wish)

```
> x                                # just arbitrary x
[1] 0.6434361 0.4814825
> Q=qr.Q(qr(A),complete=TRUE)  # still Example 3
> sum((A %*% x - y)^2)
[1] 10.10418
```


So, this is the squared distance of Ax to y. After premultiplying with $Q^\top$

```
> sum((t(Q) %*% A %*% x - t(Q) %*% y)^2)
[1] 10.10418
```

the distance remains the same. This corresponds to the fact that orthogonal transformations (multiplying by Q as well as by $Q^\top$ represents such transformations) are preserving lengths (and angles)

That is what is behind the whole thing. Of course, the full explanation is only via ...

... general mathematics!

Note that $A = QR = \dot{Q}\dot{R}$, where $Q = \begin{pmatrix} \dot{Q} & \tilde{Q} \end{pmatrix}$ is a component of the full QR decomposition and has orthonormal columns. We have

$$\|(Ax - y)\|_2^2 = (Ax - y)^\top (Ax - y) = (y - Ax)^\top (y - Ax) = \|y - Ax\|_2^2$$

$$\begin{aligned} \|(Q^\top (y - Ax)\|_2^2 &= (Q^\top (y - Ax))^\top (Q^\top (y - Ax)) \\ &= (y - Ax)^\top Q Q^\top (y - Ax) = (y - Ax)^\top (y - Ax) = \|y - Ax\|_2^2 \end{aligned}$$

$$\|Q^\top (y - Ax)\|_2^2 = \left\| \begin{pmatrix} \dot{Q}^\top \\ \tilde{Q}^\top \end{pmatrix} (y - Ax) \right\|_2^2 = \left\| \begin{pmatrix} \dot{Q}^\top y - \dot{Q}^\top \dot{Q}\dot{R}x \\ \tilde{Q}^\top y - \tilde{Q}^\top \dot{Q}\dot{R}x \end{pmatrix} \right\|_2^2$$

$$= \left\| \begin{pmatrix} \dot{Q}^\top y - \dot{R}x \\ \tilde{Q}^\top y \end{pmatrix} \right\|_2^2 = \|\dot{Q}^\top y - \dot{R}x\|_2^2 + \|\tilde{Q}^\top y\|_2^2$$

Looking for x minimizing this, we note that the second term does not depend on it, and the first term will be minimal when it become zero - that is, when $\quad \dot{R}x = \dot{Q}^\top y$

As $\dot{R}$ is $q \times q$ upper triangular, we just obtain x by backsolving - the whole thing succeds as long as $\dot{R}$ has rank q; that is true if A has full rank $\min\{p, q\}$

Conclusion

This likely explains why economy is the default in R: the whole thing serves for finding least-squares fits!

QR decomposition can nicely solve determined linear systems, even when the matrix is not square. However, it can do more: overdetermined systems, and thus, in particular, it is used to compute least-squares fits directly from the matrix X, *without a need to compute* $X^\top X$.

So, to obtain the least-squares solution b for $y \sim X\beta$:

the expert in numerical computations does not

```
solve(t(X) %*% X) %*% (t(X) %*% y)
```


but the expert in statistical computing does not even

```
solve(t(X) %*% X, t(X) %*% y)
```

or `solve(crossprod(X), crossprod(X,y))`

and not even `qr.solve(crossprod(X), crossprod(X,y))`

but `qr.solve(X, y)`