Alternatively, we can use the numpy function `np.linalg.solve(A,b)` to solve a set of linear equations

$$Ax = b.$$

This is faster than `x = np.linalg.inv(A)@ b`, which first computes the inverse of A and then multiplies it with b. However, this function computes the exact solution of a well-determined system.

In [ ]:
```
import time
n = 5000
A = np.random.normal(size = (n,n))
b = np.random.normal(size = n)
start = time.time()
x1 = np.linalg.solve(A,b)
end = time.time()
print(np.linalg.norm(b - A @ x1))
print(end - start)
```

```
4.033331000615254e-09
1.2627429962158203
```

In [ ]:
```
start = time.time()
x2 = np.linalg.inv(A) @ b
end = time.time()
print(np.linalg.norm(b - A @ x2))
print(end - start)
```

```
8.855382050136278e-10
4.3922741413116455
```

11.4. Pseudo-inverse

In Python the pseudo-inverse of a matrix `A` is obtained with `np.linalg.pinv()`. We compute the pseudo-inverse for the example of page 216 of VMLS using the `np.linalg.pinv()` function and via the formula $A^{\dagger} = R^{-1}Q^T$, where $A = QR$ is the QR factorization of A.

In [ ]:
```
A = np.array([[-3,-4],[4,6],[1,1]])
np.linalg.pinv(A)
Q, R = np.linalg.qr(A)
R
```

Out[ ]:
```
array([[ 5.09901951,  7.256297  ],
       [ 0.        , -0.58834841]])
```

In [ ]:
```
np.linalg.solve(R,Q.T)
```

Out[ ]:
```
array([[-1.22222222, -1.11111111,  1.77777778],
       [ 0.77777778,  0.88888889, -1.22222222]])
```