# Lab class (Householder reflectors and QR factorization)

## Householder reflectors

Recall that a *Householder reflector* is the matrix $H = I - \frac{2}{u^*u}uu^*$. Given a vector $x$, if we make the choice $u = x \pm \|x\|e_1$, then $Hx$ is a multiple of $e_1$ (which is the vector $\begin{bmatrix}1 & 0 & 0 & \cdots & 0\end{bmatrix}^*$). Normally we choose $u = x + \operatorname{sgn}(x_1)\|x\|e_1$, where $\operatorname{sgn}(x)$ is the sign of $x$, so that we don't have to subtract two numbers with the same sign to compute it.

The QR factorization of a square matrix $A$ can be obtained with $R = Q_{n-1} \dots Q_3 Q_2 Q_1 A$, where

$$Q_k = \begin{bmatrix} I_{(k-1)\times(k-1)} & 0 \\ 0 & H_k \end{bmatrix},$$

and $H_k \in \mathbb{C}^{(n-k+1)\times(n-k+1)}$ is a Householder reflector that transforms the first column of $A_{k:end,k:end}$ into a multiple of $e_1$.

The factor $Q$ is given by $Q = Q_1^* Q_2^* \cdots Q_{n-1}^*$ (or can be represented implicitly be the vectors $u_1, u_2, \dots, u_{n-1}$).

1. Write a `function u = householder(x)` that takes a vector $x$ and computes the Householder reflector that turns it into a multiple of $e_1$. Choose $\operatorname{sgn}(x)$ as the sign.

2. Test the function on a few vectors: compute $y = Hx$, and check that `norm(y(2:end))` is small.

   *You may use the instruction* `format('long', 'e')` *to display vectors in exponential notation, which shows a higher number of significant digits and makes it easier to display their content.*

3. Test the function on the vector given by `x = 1e-10*randn(10, 1); x(1) = 2e8`.

4. Now change your function to produce "wrong-sign Householder" with $u = x - \operatorname{sgn}(x)\|x\|e_1$, and test the function on the vector in the previous point. Check that $y$ is no closer to $e_1$ than $x$ was, in this case.

5. Use your Householder function to implement a `function R = my_qr(A)` that computes the QR factorization of a *square* matrix $A$. Don't call it `qr` or it would conflict with Matlab's built-in function, and we want to test them together later.

6. Make sure that you put parentheses correctly in your function (so that the computational cost isn't higher than necessary), and that the $R$ you return is truly upper triangular (you'll have to set the sub-diagonal entries to 0 manually, or, even better, not compute them at all...).

7. Test your function against Matlab's builtin `[Q, R] = qr(A)`. Do they return the same result?[1]

8. Check that $\|R\| = \|A\|$ (using the norm-2). Can you explain why, using the definition of norm-2 of a matrix?

You can modify later your function `my_qr` to return $Q$ as well, and to make it work for rectangular matrices, too. These points are more tricky. But first, we want to test in practice the stability of QR factorization and system solving, and for that we will use Matlab's built-in functions.

## Stability of QR factorization

*The exercises in this section are based on Lecture 16 of the Trefethen-Bau book.*

We first create a random matrix of which we know the exact QR factorization: compute

```
n = 5
R = triu(randn(n)); % random upper triangular R
[Q, dummy_variable] = qr(randn(n)); % random orthogonal Q
A = Q*R;
```

Now the factors of $A$ are $Q$ and $R$. (Almost, because of rounding errors in `Q*R`, but that is enough for our purposes. We ignore the error in matrix products, in this experiment. You can check that `norm(Q*R - A)` is small, anyway).

1. Compute the QR factorization of $A$ with `[Q2, R2] = qr(A)`. Check that `Q2` is orthogonal, that `norm(Q-Q2), norm(R-R2)` are small, and that `norm(Q2*R2 - A)` are small.

2. Now let's scale up! Change $n$ to larger values: 10, 30, 50. Put all these instructions in a script (`.m` file) so that you can re-run them more easily and test several (random) examples.

3. Notice what happens to the various error measures. You should see that `norm(Q2*R2 - A)` stays small (of the order of $10^{-15}$). Can you come up with an explanation?

   *Hint: backward stability of* `Q2, R2`.

[1] Several students in the lab noticed that their implementation returned a matrix $R_2$ which was equal to Matlab's one apart from the last entry $R_{end,end}$. This happened because of different handling of the last step: when it arrives to the last step ($1 \times 1$ reflector), Matlab simply does nothing because it is already upper triangular. If your `for` cycle goes from 1 to $n$, instead, you are applying an additional $1 \times 1$ reflector that swaps the sign of $R_{end,end}$.

4. Instead, `norm(Q-Q2), norm(R-R2)` grow wildly. Can you come up with an explanation?

   *Hint: what does this tell us about the condition number of QR factorization?*

5. We have studied only the condition number of linear system solving, but it turns out that the condition number of the QR factorization is the same, $\kappa(A) = \|A\|\|A^{-1}\|$. Try several examples, and check that the relative errors $\frac{\|Q-Q_2\|}{\|Q\|}$ and $\frac{\|R-R_2\|}{\|R\|}$ are of the order of $\kappa(A)$ times the machine precision.

   You can use `cond(A)` to return $\kappa(A)$, and `eps` to return the machine precision.

So, moral of the story: sometimes your algorithms return inaccurate results, but if you won't notice unless you have already the true result available (because `Q2*R2-A` is always small and does not reveal the errors; we needed `Q` and `R` to check). Who tells you when it's right or wrong? Condition number and error analysis!

## Matrix norms

If you still have time, you can try to check that Matlab's `norm(A)` computes the correct value using the definition of matrix norms. Recall that the norm of a matrix is defined as

$$\|A\| = \max_{v\in\mathbb{C}^n} \frac{\|Av\|}{\|v\|}.$$

1. Write a function `function M = checknorm(A, k)` that generates $k$ random vectors and for each of them computes $\frac{\|Av\|}{\|v\|}$. Return the maximum of the computed values. Note that the norms that appear in the formula are norms of vectors.

2. Choose a matrix $A$, for instance `A = randn(5)`, a large value of $k$, and test if your function returns something close to $\|A\|$ or not.

Testing vectors at random is not a great method to find the maximum of a function (and indeed it converges very slowly...). You will see with prof. Passacantando much better methods to find the maximum of a function.

(And in fact, this problem has a closed-form solution: $\|A\|^2$ is the largest eigenvalue of $A^*A$. We will prove this later when we study a different decomposition, the SVD).

## Linear system solving with QR

If you still have time, you can move on to test the stability of solving linear systems with QR. Recall that if $A = QR$, then the solution of a linear system $Ax = b$ is given by $x = R^{-1}Q^*b$. (Do you remember why?)

Matlab has a built-in operator to solve linear systems, `x = A \ b`. If we just write `R \ b`, it is smart enough to recognize that $R$ is upper triangular and solve it by backward substitution (in time $O(n^2)$). So we can use this instead of implementing our own back-substitution function (which is not complicated anyway).

1. Create a system with known solution by choosing a random $x$, a random $A$ (of the right size), and computing `b = A * x`. (As usual, we shall ignore the errors in this product.)
2. Choose a perturbation `f = 1e-8 * randn(size(b))`, and compute the solution of the linear system $A\tilde{x} = b + f$. Check that $\frac{\|\tilde{x}-x\|}{\|x\|}$ is of the order of $\kappa(A)\frac{\|f\|}{\|b\|}$.
3. If you choose `A = randn(n, n)`, you will only get matrices with small-ish condition number. Try with hand-made matrices with unbalanced entries: for instance, `A = [1000 1; 1 1]`. Or also `A = [1+1e-8 1; 1 1]`, which is very close to a singular matrix. Try several sizes of the perturbation $f$.

## Timings

If you have finished the two-output version of `[Q, R] = my_qr(A)`, you can check how fast it is with respect to Matlab's built-in one. You can measure timings in Matlab with the functions `tic` and `toc`.

```
        tic()
        [Q, R] = my_qr(A)
        time_elapsed = toc()
```

It is difficult to beat the built-in functions, since they are implemented in compiled libraries.