CSCI 2033 Assignment 3:
QR Decomposition

Posted: Saturday, November 18
Updated: Wednesday, November 22

Due date: Thursday, November 30, 11:55 pm

In this assignment, you will implement a MATLAB function to decompose a matrix $\mathbf{A}$ into the product of two matrices $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ has orthonormal columns and $\mathbf{R}$ is upper triangular. You will then use your decomposition to solve a shape fitting problem.

You will need the files `back_sub.m` for Part 4 and `generate_data.m` and `visualize.m` for Part 5. These can be found in the zip file provided on the Moodle assignment page.

1 Submission Guidelines

You will submit a zip file that contains the following `.m` files on Moodle:

- `ortho_decomp.m`
- `my_qr.m`
- `least_squares.m`
- `my_pack.m`
- `my_unpack.m`
- `design_matrix.m`
- `affine_fit.m`

as well as any helper functions that are needed by your implementation.

As with previous assignments, please note the following rules:

1. Each file must be named as specified above.
2. Each function must return the specified value(s).
3. You may use MATLAB's array manipulation syntax, e.g. `size(A)`, `A(i,:)`, and `[A,B]`, and basic operations like addition, multiplication, and transpose of matrices and vectors. High-level linear algebra functions such as `inv`, `qr`, and `A\b` are *not* allowed except where specified. Please contact the instructor with further questions.

4. This is an individual assignment, and no collaboration is allowed. Any in-person or online discussion should stop before you start discussing or designing a solution.

2 Orthogonal decomposition

Suppose you have an orthonormal basis $\{\mathbf{u}_1, \dots, \mathbf{u}_n\}$ for a subspace $H \subseteq \mathbb{R}^m$. Any vector $\mathbf{v} \in \mathbb{R}^m$ can be decomposed into the sum of two orthogonal vectors, $\hat{\mathbf{v}} \in H$ and $\hat{\mathbf{v}}^\perp \in H^\perp$. Furthermore, since $\hat{\mathbf{v}} \in H$, it can be expressed as $\hat{\mathbf{v}} = c_1\mathbf{u}_1 + \cdots + c_n\mathbf{u}_n$ for some weights $c_1, \dots, c_n$. Implement a function to perform this decomposition.

Specification:

`function [c, v_perp] = ortho_decomp(U, v)`
Input: an $m \times n$ matrix $\mathbf{U} = \begin{bmatrix}\mathbf{u}_1 & \cdots & \mathbf{u}_n\end{bmatrix}$ with orthonormal columns, and a vector $\mathbf{v}$.
Output:
`c`: a vector $\mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$ such that $\mathbf{v} = c_1\mathbf{u}_1 + \cdots + c_n\mathbf{u}_n + \hat{\mathbf{v}}^\perp$
`v_perp`: the residual vector $\mathbf{v}^\perp$ such that $\mathbf{u}_i \cdot \hat{\mathbf{v}}^\perp = 0$ for all i.

Implementation notes:

Here $\hat{\mathbf{v}}$ is simply the orthogonal projection of $\mathbf{v}$ onto the subspace H. Since we have an orthogonal basis of H, this should be easy to compute. In fact, since the basis is *orthonormal*, it is possible to compute the projection in just one line of code without any `for` loops. Hint: What are the entries of $\mathbf{U}^T\mathbf{v}$?

Test cases:

1. $\mathbf{U} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad \rightarrow \quad \mathbf{c} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \hat{\mathbf{v}}^\perp = \begin{bmatrix} 0 \\ 3 \\ 0 \\ 5 \end{bmatrix}$
2. $\mathbf{U} = \begin{bmatrix} 0.6 \\ 0.8 \\ 0 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \rightarrow \quad \mathbf{c} = \begin{bmatrix} 1.4 \end{bmatrix}, \hat{\mathbf{v}}^\perp = \begin{bmatrix} 0.16 \\ -0.12 \\ 1 \end{bmatrix}$

3. Generate a random orthonormal set of 5 vectors in $\mathbb{R}^{10}$ using the command `[U,~] = qr(randn(10,5),0)`, and generate another random integer vector `v = randi([-2,2], 10,1)`. Compute your orthogonal decomposition, `[c, v_perp] = ortho_decomp(U, v)`. Verify that `U*c + v_perp` is nearly the same as `v`, and `U'*v_perp` is nearly zero (both to within 10^{-12}).

3 QR decomposition

Suppose you have an $m \times n$ matrix $\mathbf{A}$ which is "tall and thin", i.e. with $m > n$, and the columns of $\mathbf{A}$ are linearly independent. The Gram-Schmidt process corresponds to a factorization $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is an $m \times n$ matrix with orthonormal columns, and $\mathbf{R}$ is an $n \times n$ upper triangular matrix.[1]

Specification:

`function [Q, R] = my_qr(A)`
Input: an $m \times n$ matrix $\mathbf{A}$ with $m > n$.
Output: an $m \times n$ matrix $\mathbf{Q}$ with orthonormal columns, and an $n \times n$ upper triangular matrix $\mathbf{R}$, such that $\mathbf{A} = \mathbf{QR}$.

Implementation notes:

I recommend starting your implementation by first having your function compute $\mathbf{Q}$ correctly. After that, you can modify your code to also compute $\mathbf{R}$.

The columns of $\mathbf{Q}$ are essentially obtained by performing the Gram-Schmidt process with normalization on the columns of $\mathbf{A}$:

$$
\begin{aligned}
\mathbf{q}_1 &= \mathbf{a}_1/\|\mathbf{a}_1\|, \\
\hat{\mathbf{a}}_2^{\perp} &= \mathbf{a}_2 - \operatorname{proj}_{H_1} \mathbf{a}_2 && \text{where } H_1 = \operatorname{span}\{\mathbf{q}_1\}, \\
\mathbf{q}_2 &= \hat{\mathbf{a}}_2^{\perp}/\|\hat{\mathbf{a}}_2^{\perp}\|, \\
\hat{\mathbf{a}}_3^{\perp} &= \mathbf{a}_3 - \operatorname{proj}_{H_2} \mathbf{a}_3 && \text{where } H_2 = \operatorname{span}\{\mathbf{q}_1, \mathbf{q}_2\}, \\
\mathbf{q}_3 &= \hat{\mathbf{a}}_3^{\perp}/\|\hat{\mathbf{a}}_3^{\perp}\|, \\
&\vdots
\end{aligned}
$$

You can carry this out using your orthogonal decomposition function. For each column $i = 1, \ldots, n$, call `ortho_decomp` with an appropriate set of inputs, and

[1] Technically, what we are computing here is known as the "thin" or "reduced" QR decomposition. In the full QR decomposition, $\mathbf{Q}$ is square and orthogonal, and $\mathbf{R}$ is an $m \times n$ upper triangular matrix whose lower $(m-n)$ rows are all zero. In practice, it is also not computed with the Gram-Schmidt process, but with other methods that are more numerically stable.

use the resulting $\hat{\mathbf{v}}^\perp$ to fill in the ith column of $\mathbf{Q}$. In the end, the columns of $\mathbf{Q}$ should be an orthonormal set of vectors $\{\mathbf{q}_1, \ldots, \mathbf{q}_n\}$ which span $\operatorname{Col} \mathbf{A}$.

Once you can compute $\mathbf{Q}$ correctly, modify your function to compute $\mathbf{R}$ as well. In principle, since $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, you could obtain $\mathbf{R}$ simply via $\mathbf{Q}^T\mathbf{A} = \mathbf{Q}^T\mathbf{Q}\mathbf{R} = \mathbf{R}$. However, this is more work than necessary, because you can actually fill in the entries of $\mathbf{R}$ as you compute each column of $\mathbf{Q}$. When you compute the ith column of $\mathbf{Q}$, the `ortho_decomp` call gives you both $\mathbf{c}$ and $\hat{\mathbf{v}}^\perp$; can you use these to fill in the i nonzero entries in the ith column of $\mathbf{R}$? Hint: Consider the case when $\mathbf{A}$ is already an upper triangular matrix, say $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}$, and work out by hand what $\mathbf{c}$ and $\hat{\mathbf{v}}^\perp$ will be for each column.

Test cases:

1. $\mathbf{A} = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & -4 \\ -2 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad \mathbf{Q} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{bmatrix}$
2. $\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad \rightarrow \quad \mathbf{Q} = \begin{bmatrix} 1/2 & 1/2 & -1/2 \\ 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
3. Generate a random 10×5 integer matrix `A = randi([-2,2], 10,5)` and compute your QR decomposition, `[Q, R] = my_qr(A)`. Verify that `istriu(R)` is true, `Q'*Q` is nearly the identity matrix, and `Q*R` is nearly the same as `A` (both to within 10^{-12}).

4 Least-squares problems

Use the QR decomposition to solve the least-squares problem $\mathbf{Ax} \approx \mathbf{b}$.

Specification:

`function x = least_squares(A, b)`
Input: an $m \times n$ matrix $\mathbf{A}$ and a vector $\mathbf{b} \in \mathbb{R}^m$.
Output: a vector $\mathbf{x} \in \mathbb{R}^n$ which is the least-squares solution of the over-determined system $\mathbf{Ax} \approx \mathbf{b}$, i.e. such that $\mathbf{Ax} - \mathbf{b} \in (\mathrm{Col}\,\mathbf{A})^{\perp}$.

Implementation notes:

Do not solve the normal equations $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ directly, as this is a very numerically unstable approach. Instead, compute the QR factorization of $\mathbf{A}$ and use it to solve the least-squares problem with only a matrix-vector multiplication and a back-substitution, as described in the textbook. Use the `back_sub` function provided with this assignment to perform the back-substitution.

If you cannot get your `my_qr` function to work, you may use MATLAB's built-in `qr` function here so you can still attempt Part 5. Call `qr(A,0)` to get a rectangular $\mathbf{Q}$ and square $\mathbf{R}$ as desired.

Test cases:

1. $\mathbf{A} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} \quad \rightarrow \quad \mathbf{x} = \begin{bmatrix} 2.5 \end{bmatrix}$
2. $\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 0 \\ 3 \\ 4 \\ 3 \\ 0 \end{bmatrix} \quad \rightarrow \quad \mathbf{x} = \begin{bmatrix} 0 \\ 4 \\ -1 \end{bmatrix}$
3. Generate a random 10×5 integer matrix `A = randi([-2,2], 10,5)` and a random integer vector `b = randi([-2,2], 10,1)`. Compute your least-squares solution, `x = least_squares(A, b)`, and obtain the residual `r = b - A*x`. Verify that `A'*r` is nearly zero (to within 10^{-12}).

5 Best-fitting transformations

An *affine* transformation is a combination of a linear transformation and a translation (i.e. displacement) while retaining parallelism, i.e., parallel lines remain parallel after the transformation. For example, a point $\begin{bmatrix} x \\ y \end{bmatrix}$ on A (orange square) in Figure 1 can be transformed to $\begin{bmatrix} \overline{x} \\ \overline{y} \end{bmatrix}$ on B (red parallelogram) via a linear transform $\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$, i.e.,

$$\begin{bmatrix} \overline{x} \\ \overline{y} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Note that the parallel lines stay parallel. This transform can be combined with a translation, moving the red parallelogram to blue parallelogram (C). This composite transformation can be written as:

$$\begin{bmatrix} \widetilde{x} \\ \widetilde{y} \end{bmatrix} = \begin{bmatrix} \overline{x} \\ \overline{y} \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \end{bmatrix} + \mathbf{t}, \tag{1}$$

where $\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$ is the translation vector.

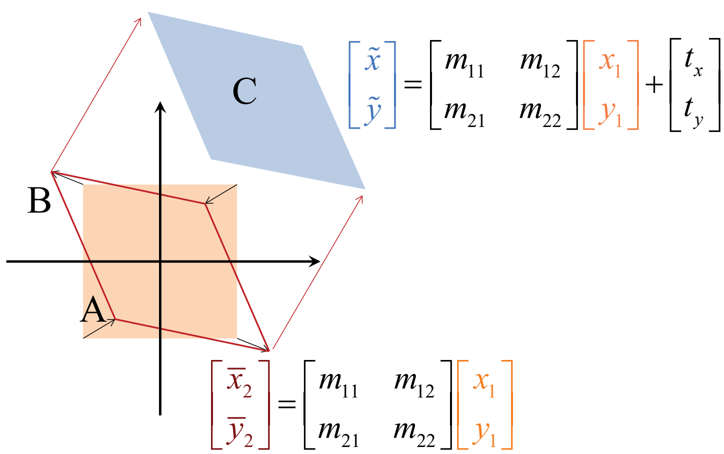


Figure 1: Affine transform.

In sum, Equation (1) can be written as

$$\begin{bmatrix} \widetilde{x} \\ \widetilde{y} \end{bmatrix} = \begin{bmatrix} m_{11}x + m_{12}y + t_x \\ m_{22}x + m_{22}y + t_y \end{bmatrix}. \tag{2}$$

5.1 Linear Equation from a Single Correspondence

Your task is given many correspondences[2] $(x_i, y_i) \leftrightarrow (\widetilde{x}_i, \widetilde{y}_i)$ where i is the index for the correspondence, to compute the best affine transform parameters, $m_{11}, m_{12}, m_{21}, m_{22}, t_x, t_y$ (unknowns): We can rewrite Equation (2) by arranging it with respect to the unknowns for, say, the first correspondence:

$$\underbrace{\begin{bmatrix} \widetilde{x}_1 \\ \widetilde{y}_1 \end{bmatrix}}_{\widetilde{\mathbf{p}}_1} = \underbrace{\begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \end{bmatrix}}_{\mathbf{A}_1} \underbrace{\begin{bmatrix} m_{11} \\ m_{12} \\ m_{21} \\ m_{22} \\ t_x \\ t_y \end{bmatrix}}_{\boldsymbol{\beta}}. \tag{3}$$

or simply,

$$\widetilde{\mathbf{p}}_1 = \mathbf{A}_1 \boldsymbol{\beta}. \tag{4}$$

Note that $\mathbf{A}_1$ depends on the original position of the point, $\mathbf{p}_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$. The correspondence produces 2 linear equations while the number of unknowns is 6, so at least 3 correspondences are needed to uniquely determine the affine transform parameters.

Implement a function `beta = my_pack(M, t)` to obtain $\boldsymbol{\beta}$ from $\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$ and $\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$, and its inverse function `[M, t] = my_unpack(beta)`.

`function beta = my_pack(M, t)`
Input: a 2×2 matrix $\mathbf{M}$ and a vector $\mathbf{t}$.
Output: a vector $\boldsymbol{\beta} \in \mathbb{R}^6$ containing the entries of $\mathbf{M}$ and $\mathbf{t}$.

`function [M, t] = my_unpack(beta)`
Input: a vector $\boldsymbol{\beta} \in \mathbb{R}^6$.
Output: a 2×2 matrix $\mathbf{M}$ and a vector $\mathbf{t} \in \mathbb{R}^2$ such that $\texttt{my\_pack}(\mathbf{M}, \mathbf{t}) = \boldsymbol{\beta}$.

Finally, construct the matrix $\mathbf{A}_i$ given $\mathbf{p}_i$.

`function Ai = design_matrix(pi)`
Input: a vector $\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \in \mathbb{R}^2$.
Output: the matrix $\mathbf{A}_i$ given by Equation (3).

[2]A point (x, y) in A is transformed to a point $(\widetilde{x}, \widetilde{y})$ in C. This pair of points forms a *correspondence*.

5.2 Solving Linear System from n Correspondences

Equation (3) can be extended to include n correspondences as follow:

$$\begin{bmatrix} \widetilde{x}_1 \\ \widetilde{y}_1 \\ \vdots \\ \widetilde{x}_n \\ \widetilde{y}_n \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 0 & 0 & 1 & 0 \\ 0 & 0 & x_n & y_n & 0 & 1 \end{bmatrix} \begin{bmatrix} m_{11} \\ m_{12} \\ m_{21} \\ m_{22} \\ t_x \\ t_y \end{bmatrix}, \tag{5}$$

or again simply,

$$\begin{bmatrix} \widetilde{\mathbf{p}}_1 \\ \vdots \\ \widetilde{\mathbf{p}}_n \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_n \end{bmatrix} \boldsymbol{\beta}, \tag{6}$$

If all correspondences are noise-free, Equation (6) will be always satisfied. Due to correspondence noise in practice, it cannot be satisfied, and therefore,

$$\begin{bmatrix} \widetilde{\mathbf{p}}_1 \\ \vdots \\ \widetilde{\mathbf{p}}_n \end{bmatrix} \approx \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_n \end{bmatrix} \boldsymbol{\beta}, \tag{7}$$

Now we will compute the best affine transform parameters using Equation (7).

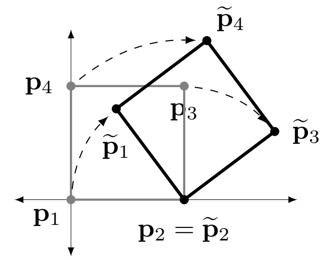
`function [M, t] = affine_fit(P, P_tilde)`
Input: $2 \times k$ correspondence matrices $\mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \cdots & \mathbf{p}_k \end{bmatrix}$ and $\widetilde{\mathbf{P}} = \begin{bmatrix} \widetilde{\mathbf{p}}_1 & \cdots & \widetilde{\mathbf{p}}_k \end{bmatrix}$ containing the reference points and transformed points as columns.
Output: a 2×2 matrix $\mathbf{M}$ and a vector $\mathbf{t} \in \mathbb{R}^2$ such that the affine transformation $\mathbf{M}\mathbf{x} + \mathbf{t}$ best matches the given data. To compute this, you will have to set up and solve the least-squares problem in Equation (7) to obtain $\boldsymbol{\beta}$, then unpack it to get $\mathbf{M}$ and $\mathbf{t}$ out.

Test cases:

For the first two tests, pick an arbitrary $\mathbf{M}$ and $\mathbf{t}$, say $\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{t} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$.

1. Check that `[M_new, t_new] = my_unpack(my_pack(M, t))` recovers the original values of $\mathbf{M}$ and $\mathbf{t}$.
2. Verify that when $\mathbf{x} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, `design_matrix(x)*my_pack(M,t)` gives back $\mathbf{t}$. Similarly, $\mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ should give you $\mathbf{m}_1 + \mathbf{t}$ and $\mathbf{m}_2 + \mathbf{t}$ respectively, where $\mathbf{m}_1$ and $\mathbf{m}_2$ are the columns of $\mathbf{M}$.

3. Applying `affine_fit` to $\mathbf{P} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$, $\widetilde{\mathbf{P}} = \begin{bmatrix} 0.4 & 1 & 1.8 & 1.2 \\ 0.8 & 0 & 0.6 & 1.4 \end{bmatrix}$ should give $\mathbf{M} = \begin{bmatrix} 0.6 & 0.8 \\ -0.8 & 0.6 \end{bmatrix}$ and $\mathbf{t} = \begin{bmatrix} 0.4 \\ 0.8 \end{bmatrix}$: a rotation and a translation.



4. We have provided a function `[P, P_tilde, M, t] = generate_data()` that produces some random test data. It does so by filling random values in $\mathbf{M}$ and $\mathbf{t}$, choosing random points $\mathbf{p}_i$, then setting each $\widetilde{\mathbf{p}}_i$ to $\mathbf{M}\mathbf{p}_i + \mathbf{t}$ plus a small amount of random noise. You can visualize the data by calling the provided function `visualize(P, P_tilde)`.

   Call `affine_fit(P, P_tilde)` to obtain your own estimates of $\mathbf{M}$ and $\mathbf{t}$. The result should be close to the "ground truth" transformation returned by `generate_data`, although due to the added noise it will not be exactly identical. Call `visualize(P, P_tilde, M, t)` to see what your fit looks like.

   Note: If you want to test on a smaller data set, just use the first few columns of $\mathbf{P}$ and $\widetilde{\mathbf{P}}$, for example `P = P(:, 1:10)` and `P_tilde = P_tilde(:, 1:10)`.