Lecture 8: Eigenvalues

Panruo Wu

NLA Chapters 24-29

Introduction

Eigenvalues are very useful in many disciplines including scientific computing and learning from data.

Example applications:

1. scientific computing
    1. vibrations (resonance & stability, e.g. 1940 Tacoma bridge collapse)
    2. cyclotrons (precise oscillations to accelerate particles).
    3. Schrodinger's equations in quantum physics/chemistry
    4. decay factors, frequencies,… in many other places
2. Data analytics
    1. spectral clustering (graph Laplacian)
    2. network link analysis (PageRank)
    3. compression, low-rank approximation

Conceptual use

Conceptually eigenvalues are useful for:

- Understanding behavior of convergence rate of iterative methods
- The local behavior of a smooth function (eigenpairs of Hessian)

Basics of Eigenvalues/Eigenvectors

For a square matrix $A \in \mathbb{R}^{m \times m}$, a nonzero vector x is an **eigenvector**, and the λ its corresponding **eigenvector,** if

$$Ax = \lambda x$$

the idea is that the action of A on some subspace mimics scalar multiplication. The subspace is called **eigenspace**

The set of all the eigenvalues are called the **spectrum** of the matrix denoted by $\Lambda(A)$.

Eigen problem only deals with square matrix, i.e. the range and domain are the same.

Eigenvalue Decomposition (EVD)

The eigenvalue decomposition can **sometimes** exist for A:

$$A = X\Lambda X^{-1}$$

where Λ is diagonal matrix with the eigenvalues; X have columns as the corresponding eigenvectors. This can be rewritten:

$$AX = X\Lambda$$

or

$$Ax_i = \lambda_i x_i, i = 1,2, \dots, m$$

EVD does not always exist. Notably it exists only when a matrix A has a full set of independent eigenvectors.

We'll give full conditions on the existence of EVD later.

Characteristic Polynomials, Multiplicity

Let us denote the eigenspace belonging to λ as E_λ. The **geometric multiplicity** is the dimension of E_λ; or equivalently the dimension of the null space of $A - \lambda I$, because $E_\lambda = \text{null}(A - \lambda I)$

Characteristic polynomial of A is denoted by

$$p_A(\lambda) = \det(\lambda I - A)$$

and the roots of $p_A(\lambda)$ coincides with all the eigenvalues of A.

The characteristic polynomial can be written with its roots:

$$p_A(\lambda) = (\lambda - \lambda_1) \cdots (\lambda - \lambda_m)$$

If an eigenvalue λ_j appears k times in the above factored form, we say eigenvalue λ_j has algebraic multiplicity k.

Similarity transformation

- A and $X^{-1}AX$ have the same characteristic polynomial, eigenvalues, algebraic/geometric multiplicity. (why?)
- The algebraic multiplicity of an eigenvalue λ is $\geq$ geometric multiplicity. (Relate basis of eigenspace to characteristic polynomial)
- **Defective eigenvalue** is an eigenvalue with a greater algebraic multiplicity than geometric multiplicity. A matrix is **defective** if it has one eigenvalue. e.g.: (which one is defective?)

$$A = \begin{bmatrix} 2 & & \\ & 2 & \\ & & 2 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{bmatrix}$$

What matrix have EVD?

If and only if it’s non-defective.

non-defective ⇔ EVD exists ⇔ diagonalizable ⇔ has m independent eigenvectors

Determinant and Trace

$$\det A = \prod_{j=1}^{m} \lambda_j \, , \mathrm{tr}(A) = \sum_{j=1}^{m} \lambda_j$$

Why? (hint: look at the characteristic polynomial)

Normal matrices

Some matrices are not only diagonalizable, they are **unitarily diagonalizable**:

$$A = Q\Lambda Q^T$$

This is both an EVD and an SVD (up to signs of Λ).

What kind of matrices are unitarily diagonalizable? One of them is the symmetric matrix. Actually, there's nice characterization for all unitarily diagonalizable matrices: normal matrice:

$$AA^T = A^T A$$

It's equivalent to saying that A has a set of orthogonal eigenvectors.

Schur Factorization

Not all matrices are diagonalizable, let alone unitarily diagonalizable… but every square matrix is unitarily triangularizable:

$$A = QTQ^T$$

where Q is orthogonal, and T is upper triangular.

So A and T have the same eigenvalues (where are they in T?)

This is called the **Schur factorization,** which turned out to be the most useful matrix factorization to reveal eigenpairs. (By now we should be familiar with the reason why we love orthogonal transformation).

Overview of Eigenvalue Algorithms

- We shall compute eigenvalues via Schur factorization.
- Eigenvalue computation is different from all matrix factorization we have studied before in that it can only be computed **iteratively.**
- We already know that finding eigenvalues through roots of characteristic polynomials is unstable (why?)
- Another idea: power iteration. What does the below sequence approach? (hint: decompose x as linear combination of eigenvectors of A).

$$\frac{x}{\|x\|}, \frac{Ax}{\|Ax\|}, \frac{A^2x}{\|A^2x\|}, \frac{A^3x}{\|A^3x\|}, \cdots$$

- However the power iteration converges pretty slowly in general, unless there's a huge gap between λ_1, λ_2.
- Most of the general purpose eigenvalue algorithms does Schur factorization via a sequence of elementary unitary similarity transformation $X \to Q_j^T X Q_j$:

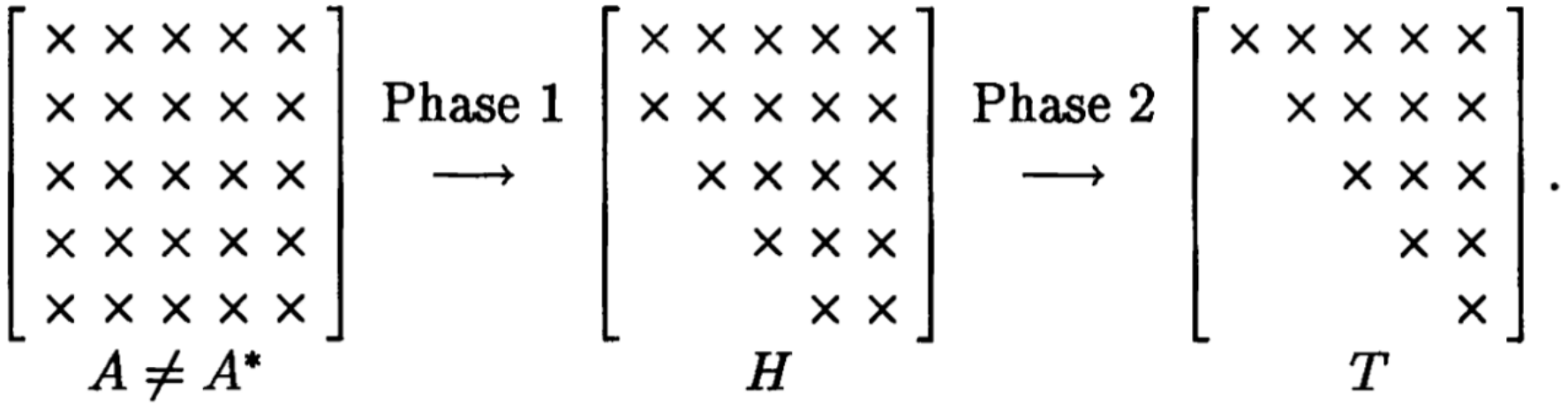
$$\underbrace{Q_j^T \cdots Q_2^T Q_1^T}_{Q^T} \; A \; \underbrace{Q_1 \cdots Q_j}_{Q}$$

converges to an upper triangular matrix T as $j \to \infty$

- The same idea as before: putting zeros into the lower half of A via orthogonal transformations (what kind of orthogonal transformation comes to mind?)

Two Phases Algorithm

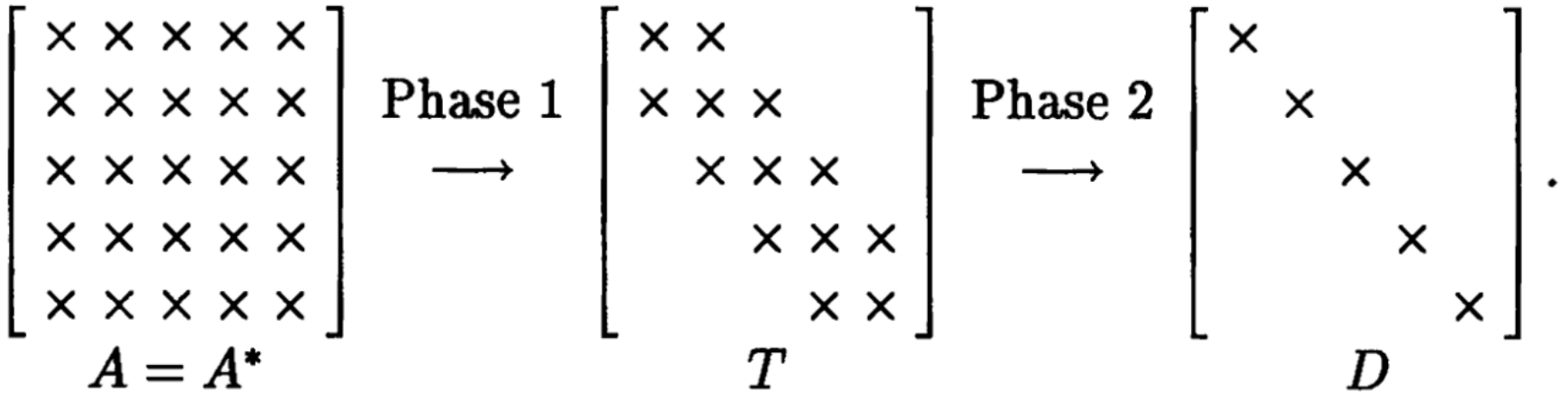
- Phase I: a direct method is applied to A to produce a upper Hessenberg matrix H (zeros below first sub-diagonal).
  - $O(m^3)$ flops
- Phase II: an iteration is applied to form an infinite sequence of Hessenberg matrices that converge to triangular form.
  - terminate in m iterations, each iterations takes $O(m^2)$ flops.



Why two phases?

Without phase I, each iteration in phase II takes $O(m^3)$ flops so in total $O(m^4)$ flops. With an upfront phase I, the total flops go down to $O(m^3)$. Phase I is clearly a profitable investment.

If A is symmetric, the phase II is even faster: $O(m^2)$ flops.



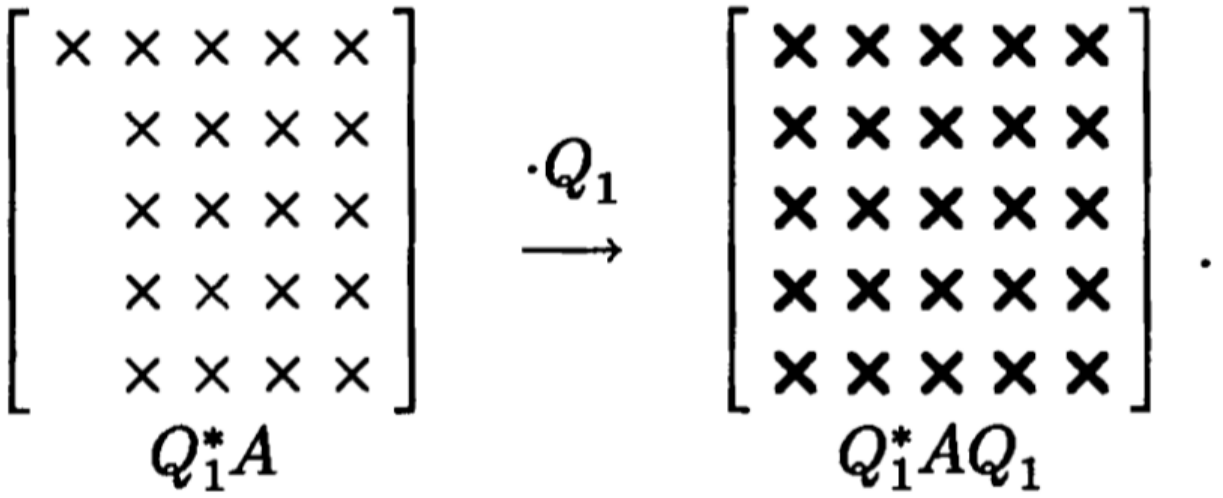
Phase I: Reduction to Hessenberg

Okay, we are back to the business of orthogonally introducing zeros to the lower part of A. At first, we are able to eliminate the column below diagonal:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{Q_1^* \cdot} \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}.$$

A $\qquad$ $Q_1^* A$

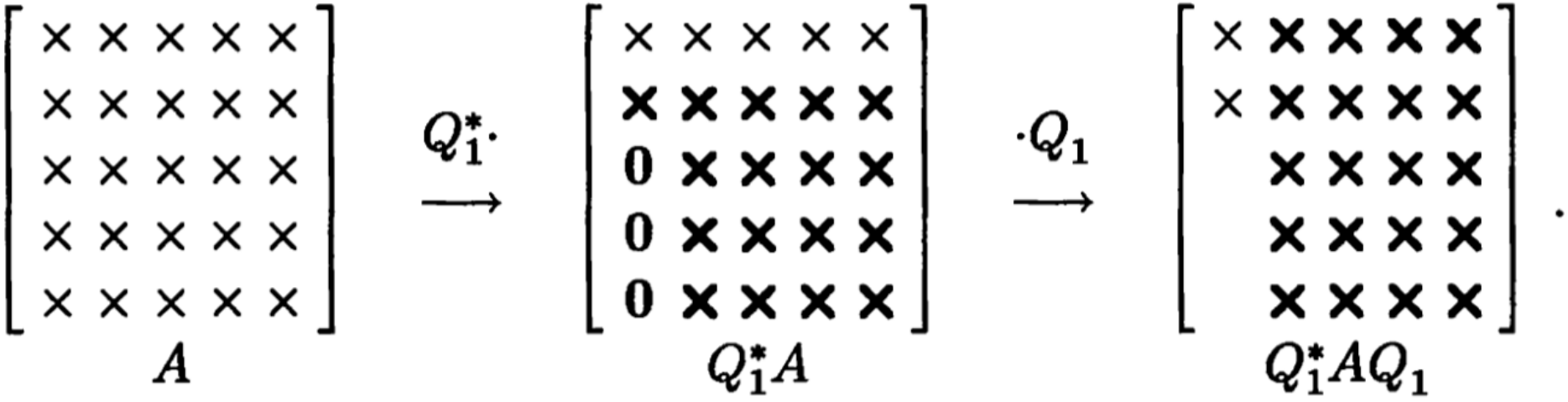
But this is a **bad** idea, because you have to symmetrically multiply Q_1 to the right:



Unfortunately, the right application of Q_1 **destroys** our newly introduced zeros…

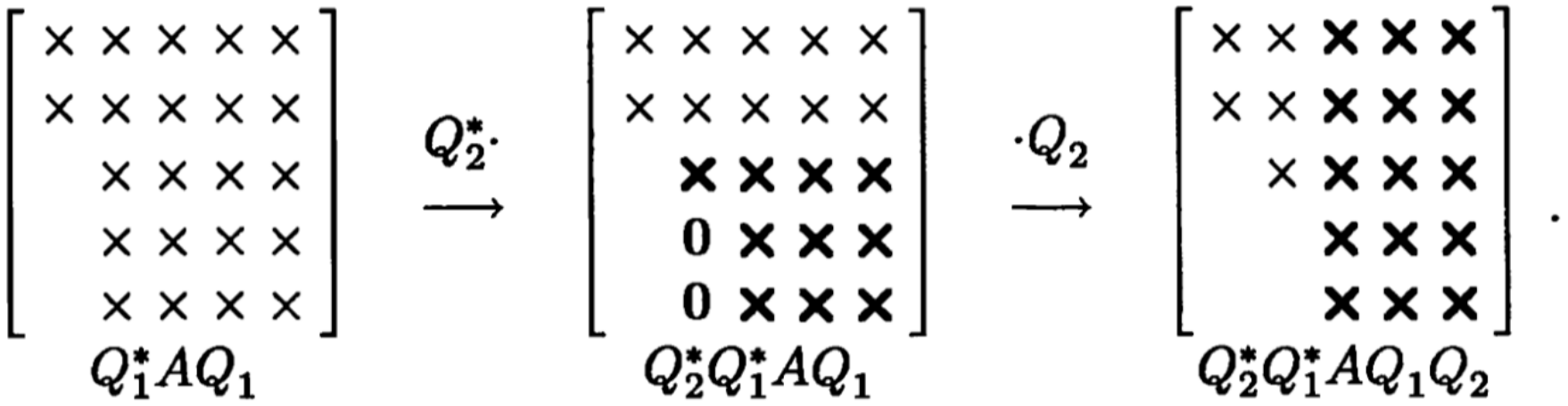
A better idea

We can't eliminate all below diagonal; let's step back and eliminate all below sub-diagonal:



It preserves the zeros! We successfully reduced **some** zeros..

Continuing this process, what do we get?



The second iteration. Eventually, we are getting:

$$\underbrace{Q_{m-2}^* \cdots Q_2^* Q_1^*}_{Q^*} A \underbrace{Q_1 Q_2 \cdots Q_{m-2}}_{Q} = H.$$

Algorithm

Algorithm 26.1. Householder Reduction to Hessenberg Form

for $k = 1$ **to** $m - 2$

$$x = A_{k+1:m,k}$$

$$v_k = \text{sign}(x_1)\|x\|_2 e_1 + x$$

$$v_k = v_k / \|v_k\|_2$$

$$A_{k+1:m,k:m} = A_{k+1:m,k:m} - 2v_k(v_k^* A_{k+1:m,k:m})$$

$$A_{1:m,k+1:m} = A_{1:m,k+1:m} - 2(A_{1:m,k+1:m} v_k)v_k^*$$

	per iteration	total
$A_{k+1:m,k:m}$	$4(m-k)^2$ flops	$\frac{4}{3}m^3$
$A_{1:m,k+1:m}$	$4(m-k)m$ flops	$2m^3$
		$\frac{10}{3}m^3$ flops

Reduction to tridiagonal

If matrix A is symmetric, you can do the same thing to make it Hessenberg and symmetric, thus tridiagonal.

And the cost is lower because of the sparsity: $\frac{4}{3}m^3$ flops.

Stability of Hessenberg/Tridiagonal reduction:

Like the HouseQR, it's backward stable if the Q is represented in the factored form (without explicitly forming Q):

$$\tilde{Q}\tilde{H}\tilde{Q}^T = A + \delta A, \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

Phase II: Hessenberg to Triangular

Phase II is necessarily an iterative process. This is where the magic happens, because we are going to devise an iterative solver that **converges cubically** (every iteration triples the correct digits). This is incredibly fast convergence rate.

But we have to study the components of QR algorithm individually first:

1. Power iteration, inverse power iteration, and Rayleigh quotient iteration
2. The QR Algorithm without/with shift

Rayleigh Quotient

We restrict ourselves to symmetric matrix (tridiagonal).

The **Rayleigh quotient** of a vector x is the quantity:

$$r(x) = \frac{x^T A x}{x^T x}$$

If x is an eigenvector, then $r(x) = \lambda$ is its eigenvalue. For an arbitrary vector x, $r(x)$ is the closest thing that acts like x's eigenvalue, in the following sense:

$$r(x) = \min_{\alpha} \|Ax - \alpha x\|_2$$

So $(x, r(x))$ kind of work like an eigenvector/eigenvalue pair.

How Rayleigh quotient approaches eigenvalue

We are interested in the asymptotic behavior of RQ as x approaches an eigenvector q_J. We claim:

$$r(x) - r(q_J) = O\left(\|x - q_J\|^2\right) \text{ as } x \to q_J$$

Why? We can do a Taylor expansion of $r(x)$ at point q_J. The first derivative:

$$\nabla r(q_J) = \frac{2}{x^T x}(Ax - r(x)x)|_{x=q_J} = 0$$

We are left with second order terms.

This is the power of Rayleigh quotient; as x approaches an eigenvector, $r(x)$ approaches its eigenvalue **quadratically.**

Power Iteration

The power iteration converges to eigenvector of the largest eigenvalue.

Algorithm 27.1. Power Iteration

$v^{(0)}$ = some vector with $\|v^{(0)}\| = 1$

for $k = 1, 2, \ldots$

$w = Av^{(k-1)}$	apply A
$v^{(k)} = w/\|w\|$	normalize
$\lambda^{(k)} = (v^{(k)})^T Av^{(k)}$	Rayleigh quotient

To analyze its convergence, write in terms of eigenvectors q_j:

$$v^0 = a_1 q_1 + \cdots + a_m q_m$$

$$v^k = c_k A^k v^0 = c_k \lambda_1^k \left(a_1 q_1 + a_2 \left(\frac{\lambda_2}{\lambda_1}\right)^k + \cdots + a_m \left(\frac{\lambda_m}{\lambda_1}\right)^k q_m\right)$$

Inverse Power Iteration

Problems of power iteration:

- It can only find the eigenvector corresponding to largest eigenvalue
- The convergence is linear, reducing error by a factor of $|\frac{\lambda_2}{\lambda_1}|$ at each iteration.

There's a way to amplify the different between the eigenvalues, by **shifting** and **inverse** the eigenvalues.

Inverse Iteration

For any μ that is not an eigenvalue, $(A-\mu I)^{-1}$ and A have the **same** eigenvectors, and the correspoinding eigenvectors become $\left\{(\lambda_j-\mu)^{-1}\right\}$. What happens if we choose μ to be very close to a specific eigenvalue λ_J? Then $(\lambda_J-\mu)^{-1}$ is much larger than any other $(\lambda_j-\mu)^{-1}$. This creates separation between the largest eigenvalue and the rest.

If we apply power iteration to $(A-\mu I)^{-1}$ then the process will quickly converge to q_J. This is called the inverse iteration.

Inverse Iteration Algorithm

- The inverse iteration can pick the eigenvector it converges to, by choosing μ to be close to its corresponding eigenvalue (if you knew it)
- The convergence is linear, but can be made much faster by creating bigger gap between the largest and second largest eigenvalues.

Algorithm 27.2. Inverse Iteration

$v^{(0)} =$ some vector with $\|v^{(0)}\| = 1$

for $k = 1, 2, \ldots$

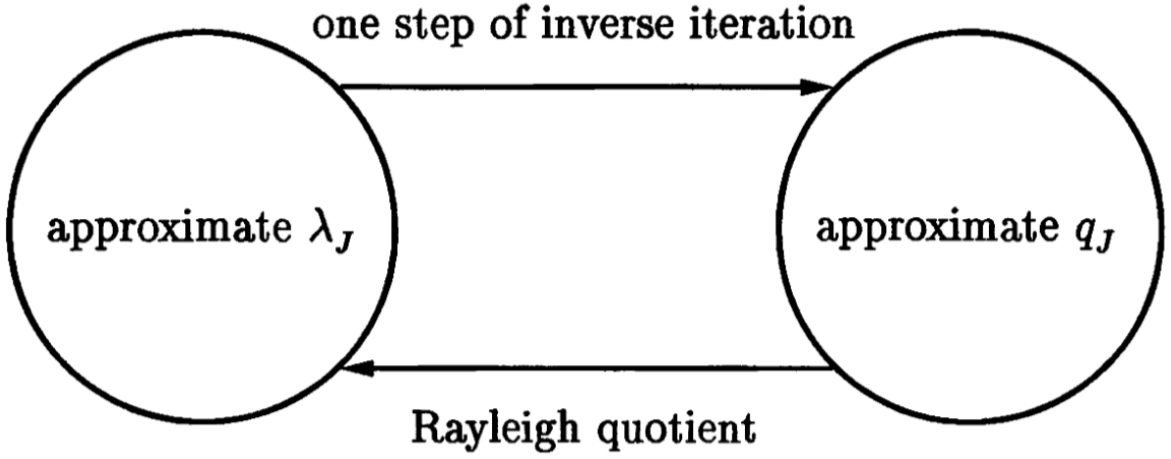
Solve $(A - \mu I)w = v^{(k-1)}$ for w	apply $(A - \mu I)^{-1}$
$v^{(k)} = w/\|w\|$	normalize
$\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$	Rayleigh quotient

Combination

So far we have two good tools at hand:

- Rayleigh quotient gives us a good eigenvalue estimate from an eigenvector estimate;
- Inverse iteration gives us a good eigenvector estimate from an eigenvalue estimate.

See how they are perfectly made for each other?



Rayleigh Quotient Iteration

Algorithm 27.3. Rayleigh Quotient Iteration

$v^{(0)} =$ some vector with $\|v^{(0)}\| = 1$

$\lambda^{(0)} = (v^{(0)})^T A v^{(0)} =$ corresponding Rayleigh quotient

for $k = 1, 2, \ldots$

Solve $(A - \lambda^{(k-1)} I) w = v^{(k-1)}$ for w	apply $(A - \lambda^{(k-1)} I)^{-1}$
$v^{(k)} = w / \|w\|$	normalize
$\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$	Rayleigh quotient

Convergence

The Rayleigh quotient iteration converges to an eigenpair for almost all starting vector v^0. The it converges, the ultimate convergence rate is cubic in that if λ_J is an eigenvalue and v^0 is sufficiently close to eigenvector q_J, then

$$\left\|v^{k+1} - \pm q_J\right\| = O\left(\left\|v^k - \pm q_J\right\|^3\right)$$

(so every iteration *triples* the number of correct digits). Why?

$$\begin{array}{rcl} \|v^{(k)} - (\pm q_J)\| & & |\lambda^{(k)} - \lambda_J| \\ \epsilon & \rightarrow & O(\epsilon^2) \\ \downarrow & \swarrow & \\ O(\epsilon^3) & \rightarrow & O(\epsilon^6) \\ \downarrow & \swarrow & \\ O(\epsilon^9) & \rightarrow & O(\epsilon^{18}) \\ \vdots & & \vdots \end{array}$$

Example

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix}$$

Let $v^0 = \frac{[1,1,1]^T}{\sqrt{3}}$. Apply Rayleigh quotient iteration in three iterations:

$$\lambda^0 = 5, \lambda^1 = 5.2131 \ldots., \lambda^2 = 5.214319743184 \ldots$$

We got 10 correct digits! This is incredible fast convergence rate. But note that this cubic convergence only happens when the v gets close enough to an eigenvector).

The QR Algorithm

Now that we have this incredibly quickly convergent Rayleigh Quotient Iteration that will give us… one eigenpair?

What if we need all of the eigenpairs? Can we do "simultaneous iterations"?

In fact we can, and the result is the jewel of numerical analysis: the QR algorithm. The pure QR algorithm is deceptively simple:

Algorithm 28.1. "Pure" QR Algorithm

$A^{(0)} = A$

for $k = 1, 2, \ldots$

$Q^{(k)} R^{(k)} = A^{(k-1)}$	QR factorization of $A^{(k-1)}$
$A^{(k)} = R^{(k)} Q^{(k)}$	Recombine factors in reverse order

“Pure” QR Algorithm

- The “pure” QR algorithm converges to an upper triangular(!!) matrix with eigenvalues on the diagonal. If furthermore A is symmetric then it converges to an diagonal matrix.
- Let’s again keep our discussion simple by assuming that A is symmetric so QR algorithm will converge to diagonal.
- What does the pure QR algorithm do? In fact, each iteration does the following:

$$R^{(k)} = \left(Q^{(k)}\right)^T A^{(k-1)}$$
$$A^{(k)} = \left(Q^{(k)}\right)^T A^{(k-1)} Q^{(k)}$$

- This is exactly the “bad” idea when we try to directly triangularize using Householder reflectors! It does not give us the upper triangular matrix we want but it *does* reduce the lower-triangular elements.

Example of pure QR algorithm

Matlab code

```
D = diag([4 3 2 1]);
rand('seed',0);
format short e
S=rand(4);
S=(S-0.5)*2;
A=S*D/S;

for i=1:20
    [Q,R] = qr(A);
    A = R*Q
end
```

Result:

```
A[1]
A =

   5.9284e+00   1.6107e+00   9.3153e-01  -2.2056e+01
  -1.5294e+00   1.8630e+00   2.0428e+00   6.5900e+00
   1.9850e-01   2.5660e-01   1.7088e+00   1.2184e+00
   2.4815e-01   1.5265e-01   2.6924e-01   4.9975e-01

A[19]
A =

   4.0000e+00   2.1427e-02  -7.0424e+00  -2.1898e+01
  -5.2787e-04   2.9997e+00   8.7369e-01  -3.1994e+00
   1.1535e-06   2.8265e-04   2.0002e+00   3.6421e+00
   4.2251e-12   7.9321e-10   2.9369e-06   1.0000e+00

A[20]
A =

   4.0000e+00   2.0896e-02  -7.0425e+00   2.1898e+01
  -3.9562e-04   2.9998e+00   8.7266e-01   3.2019e+00
   5.7679e-07   1.8846e-04   2.0002e+00  -3.6424e+00
  -1.0563e-12  -2.6442e-10  -1.4682e-06   1.0000e+00
```

Convergence rate?

```
A(20)./A(19) =  [  1.0000   0.9752   1.0000  -1.0000]
                [  0.7495   1.0000   0.9988  -1.0008]
                [  0.5000   0.6668   1.0000  -1.0001]
                [ -0.2500  -0.3334  -0.4999   1.0000]
```

QR Algorithm will iteratively eliminate the lower triangular part.

Assume that the eigenvalues of A are distinct and ordered like:

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

Then the lower triangular part converges to zero like:

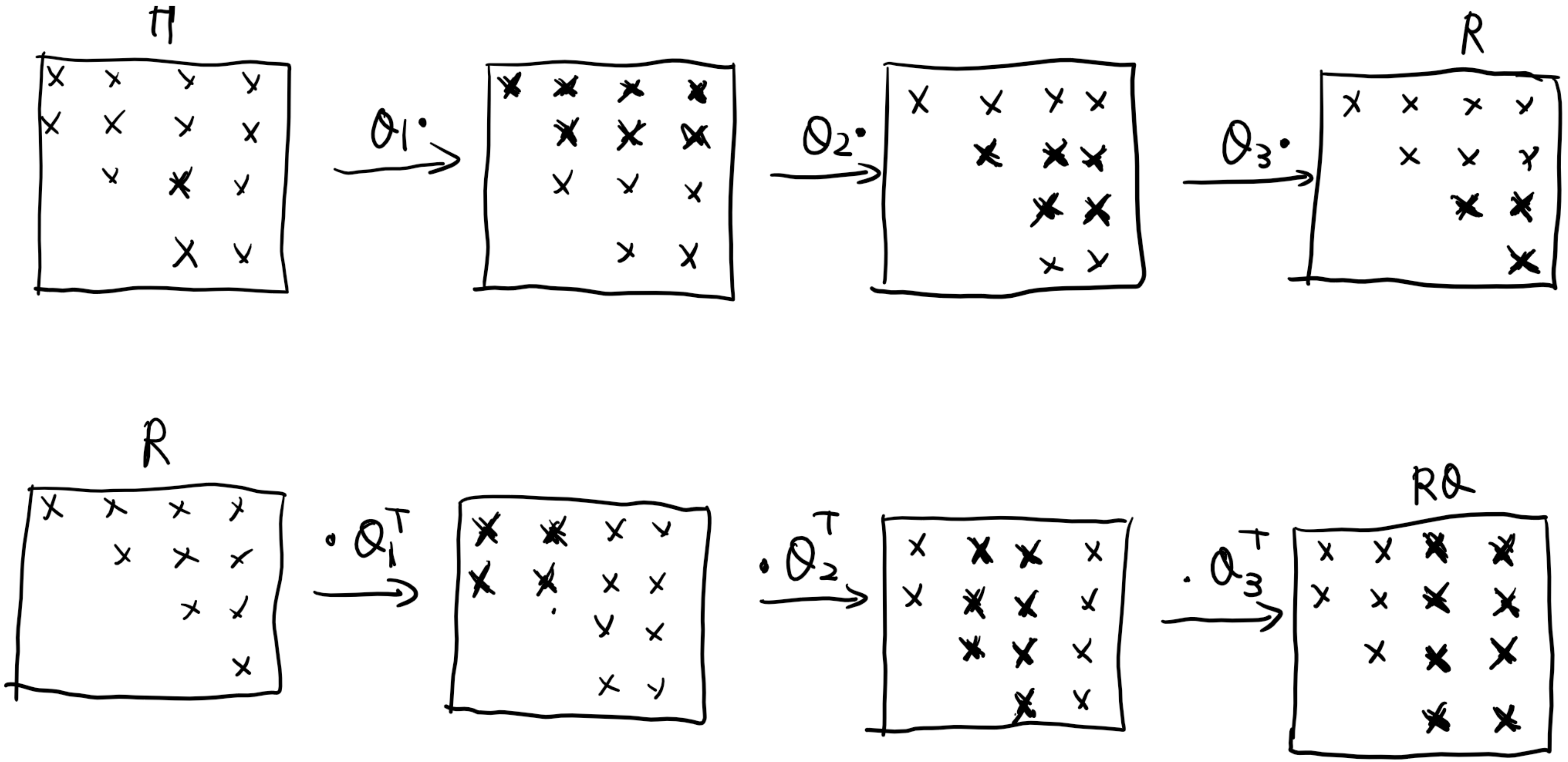
$$\left|a_{i,j}^k\right| = O\left(\left|\frac{\lambda_i}{\lambda_j}\right|^k\right), i > j$$

Two questions:

1) What is the cost of QR algorithm? (=> Phase I Hessenberg)
2) Is this convergence rate fast? (=> shifting)

The purpose of Phase I

- Phase I cost $\frac{10}{3}m^3$ flops. It gives us a Hessenberg matrix H.
- We run QR algorithm on H instead of full A. What's the cost in flops for each QR iteration?
  - The cost of QR factorization on H is $O(m^2)$ instead of $O(m^3)$ using 2x2 Householder reflectors.
  - RQ maintains the Hessenberg form? Yes!


H
$Q_1 \cdot$
$Q_2 \cdot$
$Q_3 \cdot$
R
R
$\cdot Q_1^T$
$\cdot Q_2^T$
$\cdot Q_3^T$
RQ

Improving the convergence rate

Shifting. Let's look at the following scenario: (assuming λ is an eigenvalue of H)

1. $H - \lambda I = QR$
2. $H^+ \leftarrow RQ + \lambda I$

First, we note that H^+ is similar to H. In fact

$$H^+ = Q^T(H - \lambda I)Q + \lambda I = Q^T H Q,$$

Second, because $H - \lambda I = QR$ is singular, the last entry in R is zero. Thus, the last row of RQ is also zero:

$$RQ = \begin{bmatrix} \ddots & \\ 0 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} \ddots & \\ & 0 \end{bmatrix}$$

Therefore $H^+ = RQ + \lambda I = \begin{bmatrix} H_1^+ & h_1 \\ 0 & \lambda \end{bmatrix}$. The system decouples.

Example

```
D = diag([4 3 2 1]);
rand('seed', 0);
A = S*D/S;
format short e
H = hess(A)
[Q,R] = qr(H-2*eye(4))
H1 = R*Q + 2*eye(4)
format long
lam = eig(H1(1:3,1:3))
```

```
H =
   -4.4529e-01   -1.8641e+00   -2.8109e+00    7.2941e+00
    8.0124e+00    6.2898e+00    1.2058e+01   -1.6088e+01
             0    4.0087e-01    1.1545e+00   -3.3722e-01
             0             0   -1.5744e-01    3.0010e+00

Q =
   -2.9190e-01   -7.6322e-01   -4.2726e-01   -3.8697e-01
    9.5645e-01   -2.3292e-01   -1.3039e-01   -1.1810e-01
             0    6.0270e-01   -5.9144e-01   -5.3568e-01
             0             0   -6.7130e-01    7.4119e-01

R =
    8.3772e+00    4.6471e+00    1.2353e+01   -1.7517e+01
             0    6.6513e-01   -1.1728e+00   -2.0228e+00
             0             0    2.3453e-01   -1.4912e+00
             0             0             0   -1.9099e-14

H1 =
    3.9994e+00   -3.0986e-02    2.6788e-01   -2.3391e+01
    6.3616e-01    1.1382e+00    1.9648e+00   -9.4962e-01
             0    1.4135e-01    2.8623e+00   -1.2309e+00
             0             0    1.2821e-14    2.0000e+00

lam =
   1.000000000000001,      4.000000000000013,      3.000000000000002
```

Practical QR Algorithm

But the "pure" QR algorithm is not cubically convergent. To make it so, we must combine some ideas from the Rayleigh quotient iteration, namely the shifting:

Algorithm 28.2. "Practical" QR Algorithm

$(Q^{(0)})^T A^{(0)} Q^{(0)} = A$ $\quad$ $A^{(0)}$ is a tridiagonalization of A

for $k = 1, 2, \ldots$

Pick a shift $\mu^{(k)}$ $\quad$ e.g., choose $\mu^{(k)} = A_{mm}^{(k-1)}$

$Q^{(k)} R^{(k)} = A^{(k-1)} - \mu^{(k)} I$ $\quad$ QR factorization of $A^{(k-1)} - \mu^{(k)} I$

$A^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I$ $\quad$ Recombine factors in reverse order

If any off-diagonal element $A_{j,j+1}^{(k)}$ is sufficiently close to zero,

set $A_{j,j+1} = A_{j+1,j} = 0$ to obtain

$$\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} = A^{(k)}$$

and now apply the QR algorithm to A_1 and A_2.

Example with simple shift H_{nn}

```
D = diag([4 3 2 1]);
rand('seed', 0);
S=rand(4); S = (S-0.5)*2;
A = S*D/S;
format short e
H = hess(A)

for i=1:8
    [Q,R] = qr(H-H(4,4)*eye(4));
    fprintf("H[%d]",i);
    H = R*Q + H(4,4)*eye(4)
end
```

```
H[1]
H =

   3.0067e+00   1.6742e+00  -2.3047e+01  -4.0863e+00
   5.2870e-01   8.5146e-01   1.1660e+00  -1.5609e+00
            0  -1.7450e-01   3.1421e+00  -1.1140e-01
            0            0  -1.0210e-03   2.9998e+00

H[2]
H =

   8.8060e-01  -4.6537e-01   9.1630e-01   1.6146e+00
  -1.7108e+00   5.3186e+00   2.2839e+01  -4.0224e+00
            0  -2.2542e-01   8.0079e-01   5.2445e-01
            0            0  -1.1213e-07   3.0000e+00

H[3]
H =

   1.5679e+00   9.3774e-01   1.5246e+01   1.2703e+00
   1.3244e+00   2.7783e+00   1.7408e+01   4.1764e+00
            0   3.7230e-02   2.6538e+00  -7.8404e-02
            0            0   8.1284e-15   3.0000e+00

H[4]
H =

   9.9829e-01  -7.5537e-01  -5.6915e-01   1.9031e+00
  -3.2279e-01   5.1518e+00   2.2936e+01  -3.9104e+00
            0  -1.6890e-01   8.4993e-01   3.8582e-01
            0            0  -5.4805e-30   3.0000e+00
```

```
H[5]
H =

   9.3410e-01  -3.0684e-01   3.0751e+00  -1.2563e+00
   3.5835e-01   3.5029e+00   2.2934e+01   4.1807e+00
            0   3.2881e-02   2.5630e+00  -7.2332e-02
            0            0   1.1313e-59   3.0000e+00

H[6]
H =

   1.0005e+00  -8.0472e-01  -8.3235e-01   1.9523e+00
  -7.5927e-02   5.1407e+00   2.2930e+01  -3.8885e+00
            0  -1.5891e-01   8.5880e-01   3.6112e-01
            0            0 -1.0026e-119   3.0000e+00

H[7]
H =

   9.7303e-01  -6.4754e-01  -8.9829e-03  -1.8034e+00
   8.2551e-02   3.4852e+00   2.3138e+01   3.9755e+00
            0   3.3559e-02   2.5418e+00  -7.0915e-02
            0            0  3.3770e-239   3.0000e+00

H[8]
H =

   1.0002e+00  -8.1614e-01  -8.9331e-01   1.9636e+00
  -1.8704e-02   5.1390e+00   2.2928e+01  -3.8833e+00
            0  -1.5660e-01   8.6086e-01   3.5539e-01
            0            0            0   3.0000e+00
```

- H(4,3) quadratic convergence to 0. If H is not Hessenberg but tridiagonal, the convergence rate is cubic.
- When H(4,3) is sufficiently small, we declare it to be zero, and claim we have found an eigenvalue 3.0.
- Now we continue to find other m-1 eigenvalues.

Why does QR algorithm work?

QR algorithm is related to a more obvious method called **simultaneous iteration,** where you do power iteration on not only one vector but on several simultaneously.

Let's say we have n vectors $V_0 = [v_1^{(0)}, \dots, v_n^{(n)}]$. Most likely $A^k v_1^{(0)} \rightarrow q_1$, the largest eigenvector. By similar arguments, we could see that the subspace $< A^k v_1^{(0)}, \dots, A^k v_n^{(n)} >$ converges to the subspace $< q_1, q_2, \dots, q_n >$.

If we take QR factorization of $A^k V_0 = Q^{(k)} R^{(k)}$, we could expect $Q^{(k)}$ to converge to $Q = [q_1, q_2, \dots, q_n]$.

But $A^k V_0$ is becoming increasingly ill-conditioned as $k \rightarrow \infty$. To make it feasible we can orthonormalize $A^k V_0$ in every iteration (remember what QR factorization does, in terms of subspace?)

Simultaneous Iteration

Algorithm 28.3. Simultaneous Iteration

Pick $\hat{Q}^{(0)} \in \mathbb{R}^{m \times n}$ with orthonormal columns.

for $k = 1, 2, \ldots$

$$Z = A\hat{Q}^{(k-1)}$$

$$\hat{Q}^{(k)}\hat{R}^{(k)} = Z \qquad \text{reduced QR factorization of } Z$$

Simultaneous iterations (with $V_0 = I$) is actually equivalent to QR algorithm.

Convergence and Stability

Without shifting, the QR algorithm has linear convergence (constant factor of reduction in error). With good shifting it could be cubic convergence.

Both the phase I and II are backward stable. For the symmetric eigenproblem, this means that the diagonal result satisfy:

$$\tilde{Q}\tilde{\Lambda}\tilde{Q}^T = A + \delta A, \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

Combined with the condition of symmetric eigenproblem, the forward error satisfy:

$$\frac{\left|\tilde{\lambda_j} - \lambda_j\right|}{\|A\|} = O(\epsilon_{\text{machine}})$$

SVD computation

The computation of SVD of A is based on EVD on the *covariance matrix* $A^T A$:

$$A = U\Sigma V^T$$
$$A^T A = V\Sigma^2 V^T$$

But explicitly forming $A^T A$ squared your condition number, thus making the process unstable. (Remember normal equation?)

Instead of computing on $A^T A$, there's an alternative that does not increase the condition number:

$$H = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$$

The SVD is based on implicit EVD on H, which is stable.

References

- Numerical Linear Algebra, Trefethen & Bau
- Matrix Computations, 4th edition.
- Lecture Notes on Solving Large Scale Eigenvalue Problems, Prof. Peter Arbenz. Spring 2016