UPPSALA
UNIVERSITET

Department of Information Technology

Scientific Computing for Data Analysis

Davoud Mirzaei

Lecture 10: Eigenvalues and Eigenvectors

Agenda

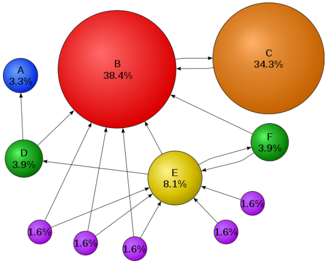
- Power method
- Real Schur decomposition
- QR-iteration

Eigenvalue computations

- How do we compute eigenvalues and eigenvectors on computers (for larger problems)?
- Going via
  $$p(\lambda) = \det(A - \lambda I)$$
  is a no-no. Way too expensive and unstable. Usually we use the opposite side to compute the roots of a polynomial!
- In some cases we only need the largest and/or smallest eigenvalues (or just a few eigenvalues) - it might be overkill to always calculate all eigenvalues:
  Example: $\|A\|_2 = \sqrt{\lambda_{max}(A^T A)}$
  Example: Steady-state of a time homogeneous Markov process is the leading eigenvector of the transition matrix
  Example: PageRank vector: leading eigenvector of Google matrix
- SVD is based on eigenvalues and eigenvectors

Example: PageRank Algorithm

- PageRank is used by Google Search to rank web pages in their search engine results.
- It is named after both the term “web page” and co-founder Larry Page
- PageRank is a way of measuring the importance of website pages.
- It works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.



Source: Wikipedia

Google Matrix

Google matrix is used by Google's PageRank algorithm. Assuming there are N pages:

- Define matrix A by $A_{i,j} = 1$ if page j has a link to page i and $A_{i,j} = 0$ otherwise
- Define matrix S from A by dividing the elements of each column by norm 1 of that column.
- If a column is zero (corresponding to dangling nodes), replace it by a constant columns with values $1/N$. It means add a link from ever dangling page.
- S is called a column stochastic matrix and belongs to the class of Markov chain operators
- Finally the Google matrix G is defined as

$$G_{ij} = \alpha S_{ij} + (1-\alpha)\frac{1}{N}, \quad 0 < \alpha < 1$$

- We can show that G has maximum eigenvalue $\lambda_1 = 1$. The normalized eigenvector v_1 gives the weights of all N pages.

Iterative methods

The only option in practice are iterative methods:

- Start with an initial value (guess)

  $$\boldsymbol{v}^{(0)}$$

  as an approximation (perhaps it is not a good approximation)
- Generate a sequence of approximations,

  $$\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \boldsymbol{v}^{(3)}, \ldots$$

- If the sequence converges, the values gets closer and closer to the real value (otherwise it diverges)
- Each step is based on the value in previous step
- The iterations are stopped when the values are good enough (the norm < chosen tolerance)

There are iterative methods for linear system of equations, nonlinear equations, eigenvalues/eigenvectors, optimization, ... For example Newton iteration for solving $f(x) = 0$:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad k = 0, 1, 2, \ldots$$

The Power Method

Let's start with an example:

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 2 \\ 1 & 3 & 1 \end{bmatrix}$$

Eigenvalues of A are

$$\lambda_1 \doteq 4.0514, \quad \lambda_2 \doteq -1.5341, \quad \lambda_3 \doteq 0.4827$$

Eigenvectors are

$$\boldsymbol{v}_1 \doteq \begin{bmatrix} -0.3873 \\ -0.5908 \\ -0.7078 \end{bmatrix}, \ \boldsymbol{v}_2 \doteq \begin{bmatrix} 0.5112 \\ -0.6477 \\ 0.5650 \end{bmatrix}, \ \boldsymbol{v}_3 \doteq \begin{bmatrix} -0.8928 \\ 0.2309 \\ 0.3867 \end{bmatrix}$$

The power method is used to approximate the leading eigenpair $\boldsymbol{v}_1$ and λ_1 (Leading eigenvalue is the largest one in magnitude)

We start the iteration with initial vector $\boldsymbol{v}^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ for example.

The Power Method

Iterations:

$$\boldsymbol{v}^{(1)} = A\boldsymbol{v}^{(0)} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}, \quad normalize \;\; \boldsymbol{v}^{(1)} = \frac{\boldsymbol{v}^{(1)}}{\|\boldsymbol{v}^{(1)}\|_2} \doteq \begin{bmatrix} 0.4243 \\ 0.5657 \\ 0.7071 \end{bmatrix}$$

$$\boldsymbol{v}^{(2)} = A\boldsymbol{v}^{(1)} \doteq \begin{bmatrix} 1.5556 \\ 2.4042 \\ 2.8284 \end{bmatrix}, \quad normalize \;\; \boldsymbol{v}^{(2)} \doteq \begin{bmatrix} 0.3865 \\ 0.5973 \\ 0.7027 \end{bmatrix}$$

$$\boldsymbol{v}^{(3)} = A\boldsymbol{v}^{(2)} \doteq \begin{bmatrix} 1.5811 \\ 2.3893 \\ 2.8812 \end{bmatrix}, \quad normalize \;\; \boldsymbol{v}^{(3)} \doteq \begin{bmatrix} 0.3891 \\ 0.5880 \\ 0.7091 \end{bmatrix}$$

Continue like this we get ...

$$\boldsymbol{v}^{(10)} \doteq \begin{bmatrix} 0.3872 \\ 0.5908 \\ 0.7078 \end{bmatrix}, \quad \boldsymbol{v}^{(11)} \doteq \begin{bmatrix} 0.3873 \\ 0.5908 \\ 0.7078 \end{bmatrix}, \quad \boldsymbol{v}^{(12)} \doteq \begin{bmatrix} 0.3873 \\ 0.5908 \\ 0.7078 \end{bmatrix}$$

Convergence (no digits changing, 4 correct decimal places)
Seems like it converges to a multiple of the leading eigenvector $\boldsymbol{v}_1$

The Power Method

To get corresponding eigenvalue one can use the so called Rayleigh quotient

$$\lambda = \frac{\boldsymbol{v}^T A \boldsymbol{v}}{\boldsymbol{v}^T \boldsymbol{v}}$$

Here we get

$$\lambda^{(12)} \doteq \begin{bmatrix} 0.3873 & 0.5908 & 0.7078 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 2 \\ 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 0.3873 \\ 0.5908 \\ 0.7078 \end{bmatrix} \doteq 4.0514$$

which is correct at least to 4 decimal places.

Note: since $\boldsymbol{v}^{(k)}$ are normalized the denominator is 1 and $\lambda = \boldsymbol{v}^T A \boldsymbol{v}$.

Note: Computation of $\lambda^{(k)}$ at each iteration does not add a new cost because the matrix-product $A\boldsymbol{v}^{(k)}$ is repeated in next iteration.

Basic structure of the power method

```
$Inputs:\ matrix\ A,\ and\ initial\ vector\ \boldsymbol{v}^{(0)}$
$Normalize\ \boldsymbol{v}^{(0)},\ and\ set\ k = 0$
while $not\ converged$
    $compute\ \boldsymbol{v}^{(k+1)} = A\boldsymbol{v}^{(k)}$
    $normalize\ \boldsymbol{v}^{(k+1)} = \frac{\boldsymbol{v}^{(k+1)}}{\|\boldsymbol{v}^{(k+1)}\|_2}$
    $compute\ \lambda^{(k+1)} = \boldsymbol{v}^{(k+1)T} A \boldsymbol{v}^{(k+1)}$
    $k = k + 1$
endwhile
```

Stop the iteration when $k > k_{max}$ or the eigenvalue or eigenvector is "good enough", typically when

$$\frac{|\lambda^{(k+1)} - \lambda^{(k)}|}{|\lambda^{(k)}|} \leq (some\ predefined\ tolerance)$$

The Power Method

- The power method ideally converges to the dominant eigenpair, i.e., $(\lambda_1, \boldsymbol{v}_1)$. (why?)
- We say λ_1 is a dominant eigenvector if

$$|\lambda_1| > |\lambda_2| \geqslant \cdots \geqslant |\lambda_{n-1}| \geqslant |\lambda_n|$$

  So λ_1 is also real.
- If substituted back (without normalization):

$$\boldsymbol{v}^{(k)} = A\boldsymbol{v}^{(k-1)} = A^2\boldsymbol{v}^{(k-2)} = \cdots = A^k\boldsymbol{v}^{(0)}$$

  Here A^k is the k-th power of A, which motivates the title "*power method*"

Some questions:

- What determines the convergence rate? $rate = \mathcal{O}(\left|\frac{\lambda_2}{\lambda_1}\right|^k)$
- What happens if $\lambda_1 = \pm\lambda_2$ while $\boldsymbol{v}_1 \neq \boldsymbol{v}_2$? convergence fails
- What happens if λ_1 is complex? convergence fails

Python code

```python
import numpy as np
def PowerMethod(A, initial_vec, tol = 1e-8, max_iter = 200):
    # Power method for computing the dominant eigenpair of matrix A
    # Inputs-  A: square matrix,   initial_vec: initial guess,
    #          tol: tolerance (default 1e-8),
    #          max_iter: maximum number of iterations (default 200)
    # Outputs- eigen_vec, eigen_val: dominant eigenpair
    #          err: error, and iter_no: number of iterations
    v_old = initial_vec
    k, lam_old, err = 0, 1, tol+1
    while k < max_iter and err >= tol:
        v_old = v_old/np.linalg.norm(v_old)
        v_new = np.matmul(A,v_old)
        lam_new = np.dot(v_old,v_new)
        err = abs(lam_old-lam_new)/abs(lam_old)
        lam_old = lam_new
        v_old = v_new
        k = k+1
    eigen_vec = v_new/np.linalg.norm(v_new)
    eigen_val = lam_new
    iter_no = k
    return eigen_vec, eigen_val, err, iter_no
```

Python code

Example: results for previous example:

```
A = np.array([[1,2,0],[1,1,2],[1,3,1]])
v0 = np.array([1,1,1])
eigen_vec, eigen_val, err, iter_no = PowerMethod(A, v0)
print('\n Egenvector = ', eigen_vec, '\n Egenvalue = ', eigen_val,
      '\n Max_iter = ', iter_no, '\n Error = ', err)
```

```
Egenvector =  [0.38725131 0.59082433 0.70778742]
Egenvalue =  4.051374236477503
Max_iter =  18
Error =  4.721300300745445e-09
```

Example: Output for matrix

$$A = \begin{bmatrix} 1 & 2 & -2 \\ -2 & 5 & -2 \\ -6 & 6 & -3 \end{bmatrix}, \qquad \begin{aligned} \lambda_1 &= 3 \\ \lambda_2 &= 3 \\ \lambda_3 &= -3 \end{aligned} \qquad V \doteq \begin{bmatrix} -0.5345 & -0.2024 & -0.3015 \\ 0.2673 & -0.7862 & -0.3015 \\ 0.8018 & -0.5838 & -0.9045 \end{bmatrix}$$

```
Egenvector =  [0.57735027 0.57735027 0.57735027]
Egenvalue =  -0.8181818181818181813
Max_iter =  200
Error =  1.454545454545455
```

Does not converge! See the `Max iter` and `Error` (Why?)

Finding the smallest eigenvalue (Inverse Power Method)

- Assume $\lambda_1, \ldots, \lambda_n$ are eigenvalues of A and

$$|\lambda_1| \geqslant \cdots \geqslant |\lambda_{n-1}| > |\lambda_n| \neq 0$$

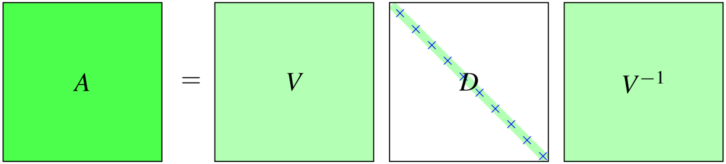
- Eigenvalues of A^{-1} are the inverse of eigenvalues of A (and eigenvectors are the same), thus $\frac{1}{\lambda_n}$ is the dominant eigenvalue of A^{-1}.
- To compute λ_n, apply the power method to A^{-1} (instead of A) and finally inverse the eigenvalue
- At step k we have

$$\boldsymbol{v}^{(k+1)} = A^{-1}\boldsymbol{v}^{(k)} \iff A\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)}$$

  Do not explicitly calculate the matrix inverse, solve the linear system instead.
- Since A is fixed in all iterations, perform an LU-factorization once, an only apply forward and backward substitution every iteration.

Similarity transform

- The idea is to find transformations similar to $A = VDV^{-1}$ and then read off the eigenvalues on the diagonal of D



- Problem: Not all matrices are diagonalizable. Instead we can use a different similarity transform
- Similarity transform:
  If $A = VBV^{-1}$ for some nonsingular matrix V, then A and B are similar $\Longrightarrow$ have the same eigenvalues
- If B happens to be triangular, then we can read off the eigenvalues from the diagonal of B (a triangular matrix has the eigenvalues on the diagonal)

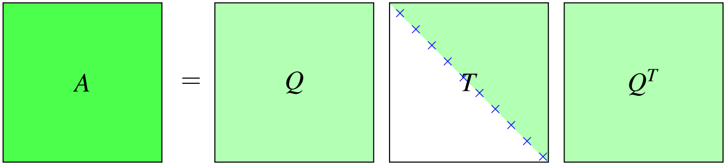
Real Schur decomposition

Real Schur decomposition (Schur form):

- If matrix A is real with real eigenvalues then there exist orthogonal matrix Q such that

$$A = QTQ^T$$

where T is real and upper triangular.



- A and T are similar $\Longrightarrow$ A's eigenvalues on the diagonal of T
- Eigenvectors? Once eigenvalues are computed, compute eigenvectors of T (by solving $(T - \lambda I)\boldsymbol{u} = 0$ which is a cheap computation as T is triangular) then eigenvector $\boldsymbol{v}$ of A is

$$\boldsymbol{v} = Q\boldsymbol{u}$$

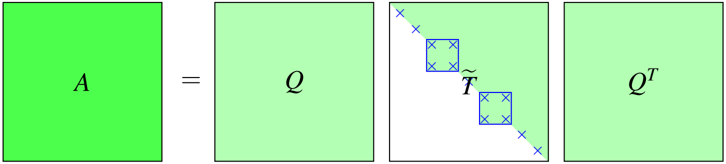
Why?

Real Schur decomposition

- If A is real but with both real and complex eigenvalues then there exist orthogonal matrix Q such that

$$A = Q\widetilde{T}Q^T$$

such that $\widetilde{T}$ is quasi upper triangular:



- Quasi upper triangular?
The little matrices has 2 eigenvalues (each) corresponding to the conjugate pair of complex eigenvalues
- If we can compute the Schur decomposition, then real eigenvalues of A are those single on-diagonals of $\widetilde{T}$, while complex eigenvalues of A are the eigenvalues of 2×2 block matrices (use the pq-formula to compute them)

Eigen-decomposition (diagonalization) vs. real Schur decomposition

$$A = VDV^{-1} \quad vs. \quad A = Q\widetilde{T}Q^T$$

- Both of them are similarity transformations: eigenvalues of A are the same as eigenvalues of D and $\widetilde{T}$
- Not all matrices are diagonalizable but Schur decomposition exists for all matrices
- Q is orthogonal ($Q^{-1} = Q^T$) while V is only non-singular
- D is diagonal but T is quasi-triangular
- When A is symmetric then $A = VDV^T$ where V is orthogonal, so eigen-decomposition is a special case of the Schur decomposition for symmetric matrices.
- Columns of V are eigenvectors of A while columns of Q are not (unless its first column)

How to compute Schur factors: QR-iteration

If we could somehow find these Schur decompositions we would have a way of finding all eigenvalues of A

- The QR-iteration is an iterative method that approaches the Schur decomposition

> - Set $A_0 = A$, "initial guess"
> - Start with finding the QR-decomposition $A_0 = Q_0 R_0$
> - Flip the order around and multiply the factors, $A_1 = R_0 Q_0$
> - QR-decomposition of A_1: $A_1 = Q_1 R_1$
> - Flip the order and multiply: $A_2 = R_1 Q_1$
> - Continue this iteration until convergence

- The QR-iteration developed by Francis in 1960's is classified as one of the topten algorithms in computation in the 20th century
- The performance and robustness is still actively improved within the numerical linear algebra research community

QR-iteration

- At step k we have

$$\begin{array}{ll} (decomposition)\ A_{k-1} = Q_{k-1}R_{k-1} & \Longrightarrow \quad R_{k-1} = Q_{k-1}^T A_{k-1} \\ (multiplication)\ \ A_k = R_{k-1}Q_{k-1} & \qquad\quad A_k = Q_{k-1}^T A_{k-1} Q_{k-1} \end{array}$$

  Thus A_k and A_{k-1} are similar. Also A_{k-1} and A_{k-2} are similar, consequently A_k and $A_0 = A$ are similar (have the same eigenvalues). We can write

$$A_k = [Q_0 Q_1 \cdots Q_{k-1}]^T A \, [Q_0 Q_1 \cdots Q_{k-1}]$$

- If we define $Q := Q_0 Q_1 \cdots Q_{k-1}$ then we have

$$A_k = Q^T A Q, \quad or \quad A = Q A_k Q^T$$

- The magic here is that A_k converges to the real Schur form $\widetilde{T}$

QR-iteration

The algorithm (in the simplest form)

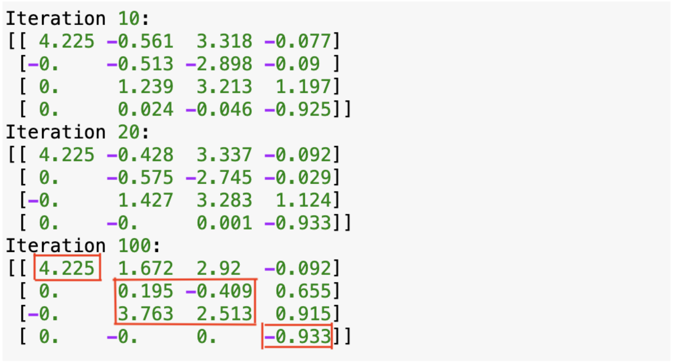
```
$Inputs:\ matrix\ A$
$set\ k = 0$
**while** $not\ converged$
    $Compute\ QR-factorization\ A = QR\ (A\ overwritten\ every\ iteration)$
    $Swap\ Q\ and\ R\ and\ multiply: A = RQ$
    $k = k + 1$
    $check\ for\ convergence$
**end while**
```

Example in Python

We use a python code to compute all eigenvalues of the following matrix using QR iteration:

$$A = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 2 & 3 & 1 & 2 \\ -1 & 0 & 2 & -1 \\ 1 & 2 & 4 & 1 \end{bmatrix}$$

Results for A_k at iterations $k = 10, 20, 100$: (edited to 3 decimals)



```
Iteration 10:
[[ 4.225 -0.561  3.318 -0.077]
 [-0.    -0.513 -2.898 -0.09 ]
 [ 0.     1.239  3.213  1.197]
 [ 0.     0.024 -0.046 -0.925]]
Iteration 20:
[[ 4.225 -0.428  3.337 -0.092]
 [ 0.    -0.575 -2.745 -0.029]
 [-0.     1.427  3.283  1.124]
 [ 0.    -0.     0.001 -0.933]]
Iteration 100:
[[ 4.225  1.672  2.92  -0.092]
 [ 0.     0.195 -0.409  0.655]
 [-0.     3.763  2.513  0.915]
 [ 0.    -0.     0.    -0.933]]
```

Example in Python

Then we compute the eigenvalues of diagonal blocks and get the list of eigenvalues: (edited to 3 decimals)

```
4.225
1.354+0.442j
1.354-0.442j
-0.933
```

The eigenvalues come in order of magnitude.
The columns of Q matrix are not identical with exact eigenvectors:

```
Q matrix =
[[ 0.387 -0.191  0.063 -0.900]
 [ 0.831 -0.226  0.279  0.425]
 [-0.295 -0.953 -0.007  0.074]
 [ 0.269 -0.071 -0.958  0.064]]

Eigenvectors =
[[ 0.844  -0.387   0.231+0.008j  0.231-0.008j]
 [-0.223  -0.831   0.650         0.650        ]
 [ 0.128   0.295  -0.449-0.048j -0.449+0.048j]
 [-0.470  -0.269  -0.542+0.16j  -0.542-0.16j ]]
```

Eigenvectors are computed using the build-in function `eig` in Python.

QR iteration for a symmetric matrix

QR iteration applied on symmetric matrix

$$A = \begin{bmatrix} 14 & -15 & -9 \\ -15 & 34 & 34 \\ -9 & 34 & 42 \end{bmatrix}$$

gives

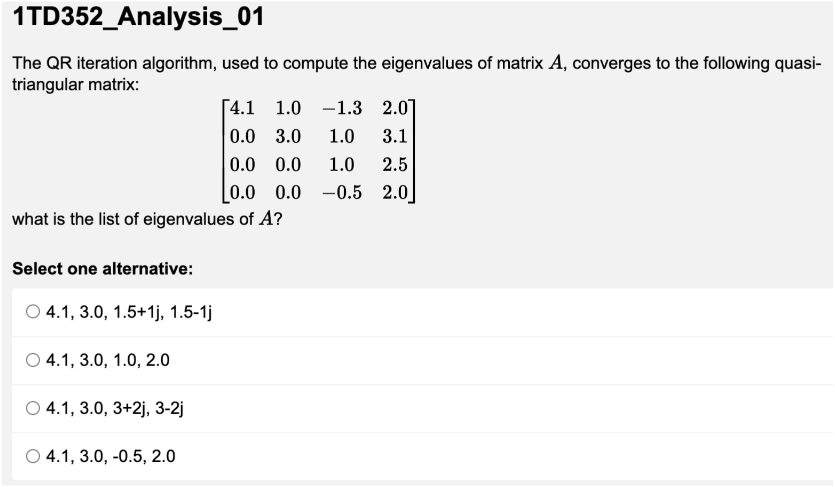
```
Iteration 10:
[[76.705 -0.     -0.    ]
 [-0.    12.475  0.    ]
 [-0.     0.      0.819]]

Q matrix =
[[ 0.259  0.834 -0.488]
 [-0.656 -0.219 -0.723]
 [-0.709  0.507  0.49 ]]

Eigenvectors =
[[-0.259  0.834 -0.488]
 [ 0.656 -0.219 -0.723]
 [ 0.709  0.507  0.49 ]]
```

The triangular matrix in indeed diagonal, and eigenvectors are identical with the columns of Q

An old exam question



1TD352_Analysis_01

The QR iteration algorithm, used to compute the eigenvalues of matrix A, converges to the following quasi-triangular matrix:

$$\begin{bmatrix} 4.1 & 1.0 & -1.3 & 2.0 \\ 0.0 & 3.0 & 1.0 & 3.1 \\ 0.0 & 0.0 & 1.0 & 2.5 \\ 0.0 & 0.0 & -0.5 & 2.0 \end{bmatrix}$$

what is the list of eigenvalues of A?

Select one alternative:

○ 4.1, 3.0, 1.5+1j, 1.5-1j

○ 4.1, 3.0, 1.0, 2.0

○ 4.1, 3.0, 3+2j, 3-2j

○ 4.1, 3.0, -0.5, 2.0

Convergence of QR-iteration

- A complete understanding of convergence of the QR method requires theories which are beyond the scope of this course.
- Just let's say, in a special case where the eigenvalues are distinct in modulus and ordered as $|\lambda_1| > |\lambda_2| \cdots > |\lambda_n|$, under certain assumptions, the elements of the matrix A_k below the diagonal will converge to zero according to

$$|a_{ij}^{(k)}| = \mathcal{O}((|\lambda_i/\lambda_j|)^k), \quad \textit{for all } i > j$$

- So, the basic QR-iteration can be slow if the eigenvalues are close to each other.

Speeding things up

Disadvantages of basic QR-iteration:

1. For a $n \times n$ matrix A, each QR-factorization costs for $\mathcal{O}(n^3)$ flops, each multiplication also $\mathcal{O}(n^3)$ flops $\Longrightarrow$ Each step costs for $\mathcal{O}(n^3)$ flops $\Longrightarrow$ If k iterations apply the total cost is $k\mathcal{O}(n^3)$
2. Usually, many steps are required to have convergence; certainly much more than n (when size of eigenvalues are close to each other). Thus the total cost is at least $\mathcal{O}(n^4)$ (inefficient)

An idea is to use a two-phase approach:

$$
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times
\end{bmatrix}
\xrightarrow{phase1}
\begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
 & \times & \times & \times & \times \\
 & & \times & \times & \times \\
 & & & \times & \times
\end{bmatrix}
\xrightarrow{phase2}
\begin{bmatrix}
\times & \times & \times & \times & \times \\
 & \times & \times & \times & \times \\
 & & \times & \times & \times \\
 & & \times & \times & \times \\
 & & & & \times
\end{bmatrix}
$$

Original A Hessenberg form H quasi-triangular

Speeding things up

- Phase 1: (before iterations) convert A to Hessenberg form using Householder reflections: $A = PHP^T$. Cost $\mathcal{O}(n^3)$
- Phase 2: (iteration starts) Apply QR iteration on H using Givens rotations. Cost at each iteration $\mathcal{O}(n^2)$ because at most $n-1$ entries are nonzero (sub-diagonals).
- The key point is that, if the basic QR-iteration is applied to a Hessenberg matrix, then all iterates A_k are Hessenberg matrices. So, the same matrix form in all iteration
- Total cost $=$ phase 1 $+$ phase 2 $= \mathcal{O}(n^3) + k\mathcal{O}(n^2)$
- In addition, different shift strategies reduce number of iterations significantly and makes it always convergent and efficient (we do not consider them here!)

House-keeping

- Mini-project 2 first submission is due Feb. 26th
- Peer review is due March 1st
- Final submissions (both miniprojects): March 8th
- Feb. 27 at 15:15 - Structure of exam and old exam questions
- Feb. 29 at 13:15 - Continue with old exam questions