4 Eigenvalues

Example 4.0.1 (Normal mode analysis).

Lecture Physik I, Section 11.2.2: equations of motion for an atom of reduced mass m in the field of another atom, using an harmonic (i.e. quadratic) approximation for the potential:

$$\ddot{x} + \omega_0^2 x = 0 \text{, with } \omega_0 = \sqrt{\frac{k}{m}} \text{, and } k = D^2 U(x^*) \text{ .}$$

In the computation of the **IR spectra of a molecule** one is interested into the vibrational frequencies of a molecule which is described by n positional degrees of freedom $\mathbf{x} \in \mathbb{R}^n$ and corresponding velocities $\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} \in \mathbb{R}^n$.

Suppose all masses are equal with an effective mass m ▶ kinetic energy $K(\mathbf{x}, \dot{\mathbf{x}}) = \frac{m}{2} \|\dot{\mathbf{x}}\|_2^2$

Model for total potential energy $U(\mathbf{x})$

1. find a local minimum $\mathbf{x}^*$ of the potential energy:

$$DU(\mathbf{x}^*) = 0\ , \quad \mathbf{H} = D^2U(\mathbf{x}^*) \text{ sym. pos. semi-def. ,}$$

hence with Taylor we have near the local minimum: find a local minimum $\mathbf{x}^*$ of the potential energy:

$$U(\mathbf{x}^* + \mathbf{a}) = U(\mathbf{x}^*) + \frac{1}{2}\mathbf{a}^T\mathbf{H}\mathbf{a}\ .$$

2. Newton'mechanics near the local minimum:

$$m\ddot{\mathbf{x}} = m\ddot{\mathbf{a}} = -D_{\mathbf{a}}U(\mathbf{x}^* + \mathbf{a})\ .$$

As we are around the minimum:

$$m\ddot{\mathbf{a}} = -D_{\mathbf{a}}(U(\mathbf{x}^*) + \frac{1}{2}\mathbf{a}^T\mathbf{H}\mathbf{a}) = -\mathbf{H}\mathbf{a}\ .$$

We obtained the equation of motion for small displacements:

$$\ddot{\mathbf{a}} + \frac{1}{m}\mathbf{H}\mathbf{a} = 0\text{, with}$$
$$\mathbf{H} = D^2U(\mathbf{x}^*)$$

3. As $\mathbf{H}$ is real and symmetric, its eigenvectors $\mathbf{w}^j$, with $j = 1, \ldots, n$ are orthogonal and hence they form a conveninent basis for representing any vector:

$$\mathbf{a} = c_1(t)\mathbf{w}^1 + \ldots + c_n(t)\mathbf{w}^n \ .$$

Inserting into the system of second-order ODEs, we get:

$$m(\ddot{c}_1\mathbf{w}^1 + \ldots + \ddot{c}_n\mathbf{w}^n) = -(c_1\mathbf{H}\mathbf{w}^1 + \ldots + c_n\mathbf{H}\mathbf{w}^n) = -(c_1\lambda_1\mathbf{w}^1 + \ldots + c_n\lambda_n\mathbf{w}^n)$$

where we denoted the associated eigenvalues by λ_j: $\mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{w}^j$.

4. Taking the scalar product with the eigenvector $\mathbf{w}^k$ we obtain an uncoupled set of equations for each $c_k(t)$, $k = 1, \ldots, n$:

$$m\ddot{c}_k = -\lambda_k c_k \ .$$

5. Looking for solutions of the form $c_k = \alpha_k \sin(\omega_k t)$ and substituting it into the differential equation one get the **angular vibrational frequency of the normal mode** $\omega_k = \sqrt{\lambda_k/m}$

6. In case of different masses we end with the system

$$\mathbf{M}\ddot{\mathbf{a}} = -\mathbf{H}\mathbf{a} \ .$$

with the mass matrix $\mathbf{M}$ which is symmetric, positiv-definite, but not necessarily diagonal. We thus must perform normal mode analysis on the matrix $\mathbf{M}^{-1}\mathbf{H}$:

$$\mathbf{M}^{-1}\mathbf{H}\mathbf{w}^j = \lambda_j \mathbf{w}^j \quad \Longleftrightarrow \quad \mathbf{H}\mathbf{w}^j = \lambda_j \mathbf{M}\mathbf{w}^j \ .$$

◇

Example 4.0.2. Physik I, Section 11.4: forced oscillation

$$\ddot{x} + 2\rho\dot{x} + \omega_0^2 x = \frac{F_0}{m}\cos\Omega t \ ,$$

has resonance for $\Omega = \omega_0\sqrt{1 - 2\rho^2/\omega_0^2}$.
The system

$$\mathbf{M}\ddot{\mathbf{x}} + 2\mathbf{B}\dot{\mathbf{x}} + \mathbf{C}\mathbf{x} = \mathbf{f} \ ,$$

with mass matrix $\mathbf{M}$, damping matrix $\mathbf{B}$, stiffness matrix $\mathbf{C}$ and force vector $\mathbf{f}$ has analogously its eigenfrequencies given by the solution of the generalized eigenvalue problem The system

$$\mathbf{C}\mathbf{x} = \omega^2 \mathbf{M}\mathbf{x} \ .$$

◇

Example 4.0.3 (Analytic solution of homogeneous linear ordinary differential equations). $\longrightarrow$ [52, Remark 5.6.1]

Autonomous homogeneous linear ordinary differential equation (ODE):

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad , \quad \mathbf{A} \in \mathbb{C}^{n,n} \; . \tag{4.0.1}$$

$$\mathbf{A} = \mathbf{S} \underbrace{\begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}}_{=:\mathbf{D}} \mathbf{S}^{-1} \; , \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regular} \quad \Longrightarrow \quad \left(\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad \overset{\mathbf{z}=\mathbf{S}^{-1}\mathbf{y}}{\longleftrightarrow} \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \right) \; .$$

➢ solution of initial value problem:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \; , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{C}^n \quad \Rightarrow \quad \mathbf{y}(t) = \mathbf{S}\mathbf{z}(t) \; , \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \; , \quad \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0 \; .$$

The initial value problem for the *decoupled* homogeneous linear ODE $\dot{\mathbf{z}} = \mathbf{D}\mathbf{z}$ has a simple analytic solution

$$\mathbf{z}_i(t) = \exp(\lambda_i t)(\mathbf{z}_0)_i = \exp(\lambda_i t) \left((\mathbf{S}^{-1})^T_{i,:} \mathbf{y}_0 \right) \; .$$

In light of Rem. **??**:

$$\mathbf{A} = \mathbf{S} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} \mathbf{S}^{-1} \quad \Leftrightarrow \quad \mathbf{A} \left((\mathbf{S})_{:,i} \right) = \lambda_i \left((\mathbf{S})_{:,i} \right) \quad i = 1, \dots, n \; . \tag{4.0.2}$$

In order to find the transformation matrix $\mathbf{S}$ all non-zero solution vectors (= eigenvectors) $\mathbf{x} \in \mathbb{C}^n$ of the linear eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

have to be found.

$\diamond$

Example 4.0.4 (Vibrating String). Vibration of a string, fixed at both ends and under uniform tension:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{T}{m(x)}\frac{\partial^2 u(x,t)}{\partial x^2} \ ,$$

with T and $m(x)$ being the tension and the mass per unit length. Separation of variables yields the problem

$$\frac{T}{m(x)}\frac{d^2 y(x)}{dx^2} + \omega^2 y(x) = 0 \ ,$$

with ω to be determined from the boundary conditions. In case of the string (one dimensional problem), we can obtain ω analyticaly, but in several diemensions not anymore.
A possibility is to use finite differences for the last differential equation ($x_i = x_1 + ih$, $y_i \approx y(x_i)$):

$$\frac{T}{m_i}\frac{y_{i-1} - 2y_i + y_{i+1}}{2h} + \omega^2 y_i = 0 \ , i = 1, 2, \ldots, N-1 \ bcom$$

which boilds down to the eigenvalue problem

$$\mathbf{A}\mathbf{y} = \omega^2 \mathbf{y} \ ,$$

with tridiagonal matrix $\mathbf{A}$ and $\mathbf{y} = (y_1, \dots, y_{N-1})^T$.

$\diamond$

4.1 Theory of eigenvalue problems

Definition 4.1.1 (Eigenvalues and eigenvectors)**.**

- $\lambda \in \mathbb{C}$ *eigenvalue* (ger.: *Eigenwert*) *of* $\mathbf{A} \in \mathbb{K}^{n,n}$ $:\Leftrightarrow$ $\underbrace{\det(\lambda \mathbf{I} - \mathbf{A})}_{\text{characteristic polynomial } \chi(\lambda)} = 0$
- *spectrum of* $\mathbf{A} \in \mathbb{K}^{n,n}$: $\sigma(\mathbf{A}) := \{\lambda \in \mathbb{C} \colon \lambda \text{ eigenvalue of } \mathbf{A}\}$
- *eigenspace* (ger.: *Eigenraum*) *associated with eigenvalue* $\lambda \in \sigma(\mathbf{A})$:
$$\mathrm{Eig}_{\mathbf{A}}(\lambda) := \mathrm{Ker}(\lambda \mathbf{I} - \mathbf{A})$$
- $\mathbf{x} \in \mathrm{Eig}_{\mathbf{A}}(\lambda) \setminus \{0\} \quad \Rightarrow \quad \mathbf{x}$ *is eigenvector*
- *Geometric multiplicity* (ger.: *Vielfachheit*) *of an eigenvalue* $\lambda \in \sigma(\mathbf{A})$:

Two simple facts:

$$\lambda \in \sigma(\mathbf{A}) \quad \Rightarrow \quad \dim \mathrm{Eig}_{\mathbf{A}}(\lambda) > 0 \,, \tag{4.1.1}$$

$$\det(\mathbf{A}) = \det(\mathbf{A}^T) \quad \forall \mathbf{A} \in \mathbb{K}^{n,n} \quad \Rightarrow \quad \sigma(\mathbf{A}) = \sigma(\mathbf{A}^T) \,. \tag{4.1.2}$$

✎ notation: $\rho(\mathbf{A}) := \max\{|\lambda|\colon \lambda \in \sigma(\mathbf{A})\} \hat{=}$ spectral radius of $\mathbf{A} \in \mathbb{K}^{n,n}$

Theorem 4.1.2 (Bound for spectral radius)**.**
For any matrix norm $\|\cdot\|$ induced by a vector norm ($\longrightarrow$ Def. 1.1.12)

$$\rho(\mathbf{A}) \leq \|\mathbf{A}\| \,.$$

Lemma 4.1.3 (Gershgorin circle theorem)**.** *For any $\mathbf{A} \in \mathbb{K}^{n,n}$ holds true*

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^{n} \{z \in \mathbb{C}\colon |z - a_{jj}| \leq \sum\nolimits_{i \neq j} |a_{ji}|\} \,.$$

Lemma 4.1.4 (Similarity and spectrum)**.**

The spectrum of a matrix is invariant with respect to similarity transformations:

$$\forall \mathbf{A} \in \mathbb{K}^{n,n}: \quad \sigma(\mathbf{S}^{-1}\mathbf{A}\mathbf{S}) = \sigma(\mathbf{A}) \quad \forall \text{ regular } \mathbf{S} \in \mathbb{K}^{n,n} \ .$$

Lemma 4.1.5. *Existence of a one-dimensional invariant subspace*

$$\forall \mathbf{C} \in \mathbb{C}^{n,n}: \quad \exists \mathbf{u} \in \mathbb{C}^{n}: \quad \mathbf{C}\left(\operatorname{Span}\{\mathbf{u}\}\right) \subset \operatorname{Span}\{\mathbf{u}\} \ .$$

Theorem 4.1.6 (Schur normal form)**.**

$$\forall \mathbf{A} \in \mathbb{K}^{n,n}: \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary}: \quad \mathbf{U}^H\mathbf{A}\mathbf{U} = \mathbf{T} \quad \text{with } \mathbf{T} \in \mathbb{C}^{n,n} \text{ upper triangular} \ .$$

Corollary 4.1.7 (Principal axis transformation)**.**

$\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$: $\exists \mathbf{U} \in \mathbb{C}^{n,n}$ *unitary*: $\mathbf{U}^H\mathbf{A}\mathbf{U} = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$, $\lambda_i \in \mathbb{C}$.

A matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ with $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$ is called normal.

Examples of normal matrices are

- Hermitian matrices: $\mathbf{A}^H = \mathbf{A}$ ➤ $\sigma(\mathbf{A}) \subset \mathbb{R}$
- unitary matrices: $\mathbf{A}^H = \mathbf{A}^{-1}$ ➤ $|\sigma(\mathbf{A})| = 1$
- skew-Hermitian matrices: $\mathbf{A} = -\mathbf{A}^H$ ➤ $\sigma(\mathbf{A}) \subset i\mathbb{R}$

➤

Normal matrices can be diagonalized by *unitary* similarity transformations

Symmetric real matrices can be diagonalized by *orthogonal* similarity transformations

In Thm. 4.1.7:
- $\lambda_1, \dots, \lambda_n$ = eigenvalues of $\mathbf{A}$
- Columns of $\mathbf{U}$ = orthonormal basis of eigenvectors of $\mathbf{A}$

Eigenvalue problems: (EVPs)

❶ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find **all** eigenvalues (= spectrum of $\mathbf{A}$).

❷ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find $\sigma(\mathbf{A})$ plus **all** eigenvectors.

❸ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find **a few** eigenvalues and associated eigenvectors

(Linear) generalized eigenvalue problem:

Given $\mathbf{A} \in \mathbb{C}^{n,n}$, regular $\mathbf{B} \in \mathbb{C}^{n,n}$, seek $\mathbf{x} \neq 0$, $\lambda \in \mathbb{C}$

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \,. \tag{4.1.3}$$

$\mathbf{x} \mathrel{\hat{=}}$ generalized eigenvector, $\lambda \mathrel{\hat{=}}$ generalized eigenvalue

Obviously every generalized eigenvalue problem is equivalent to a standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A} = \lambda \mathbf{x} \,.$$

However, usually it is not advisable to use this equivalence for numerical purposes!

Remark 4.1.1 (Generalized eigenvalue problems and Cholesky factorization).

If $\mathbf{B} = \mathbf{B}^H$ s.p.d. ($\longrightarrow$ Def. **??**) with Cholesky factorization $\mathbf{B} = \mathbf{R}^H\mathbf{R}$

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \widetilde{\mathbf{A}}\mathbf{y} = \lambda \mathbf{y} \quad \text{where } \widetilde{\mathbf{A}} := \mathbf{R}^{-H}\mathbf{A}\mathbf{R}^{-1},\ \mathbf{y} := \mathbf{R}\mathbf{x} \,.$$

→ This transformation can be used for efficient computations.

4.2 "Direct" Eigensolvers

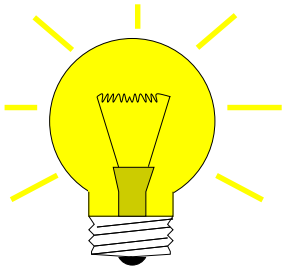
Purpose: solution of eigenvalue problems ❶, ❷ for dense matrices "up to machine precision"

python-functions: `numpy.linalg.eig`, `scipy.linalg.eig`

`w = eigvals(A)`	:	computes spectrum $\sigma(\mathbf{A}) = \{w_1, \dots, w_n\}$ of $\mathbf{A} \in \mathbb{C}^{n,n}$
`w, V = eig(A)`	:	computes spectrum w and corresponding normed eigenvectors
`eigvalsh(A)` and `eigh(A)`	:	specialized algorithms for Hermitial matrices
	$\Rightarrow$	wrappers to `lapack`-functions

Remark 4.2.1 (QR-Algorithm). → [20, Sect. 7.5]

Note: All "direct" eigensolvers are iterative methods



Idea: Iteration based on successive **unitary** similarity transformations

$$\mathbf{A} = \mathbf{A}^{(0)} \longrightarrow \mathbf{A}^{(1)} \longrightarrow \dots \longrightarrow \begin{cases} \text{diagonal matrix} & \text{, if } \mathbf{A} = \mathbf{A}^H \text{ ,} \\ \text{upper triangular matrix} & \text{, else.} \\ (\longrightarrow \text{Thm. 4.1.6}) & \end{cases}$$

(superior stability of unitary transformations, see Rem. **??**)

QR-algorithm (with shift)

- in general: quadratic convergence
- cubic convergence for normal matrices (→ [20, Sect. 7.5,8.2])

Code 4.2.2: QR-algorithm with shift

```
'''
QR-algorithm with shift
'''
from numpy import mat, argmin, eye, tril
from numpy.linalg import norm, eig, qr, eigvals

def eigqr(A,tol):
    n = A.shape[0]
    while (norm(tril(A,-1), ord=2) > tol*norm(A, ord=2)):
        # shift by ew of lower right 2×2 block closest to (A)_{n,n}
        sc, dummy = eig(A[n-2:n,n-2:n])
        k = argmin(abs(sc - A[n-1,n-1]))
        shift = sc[k]
        Q, R = qr( A - shift * eye(n))
        A = mat(Q).H*mat(A)*mat(Q);
    d = A.diagonal()
    return d

if __name__ == "__main__":
    A = mat('1,2,3; 4,5,6; 7,8,9')
    print 'numpy.linalg.eigvals:', eigvals(A)
    print 'with our eigqr::', eigqr(A,10**-6)
```

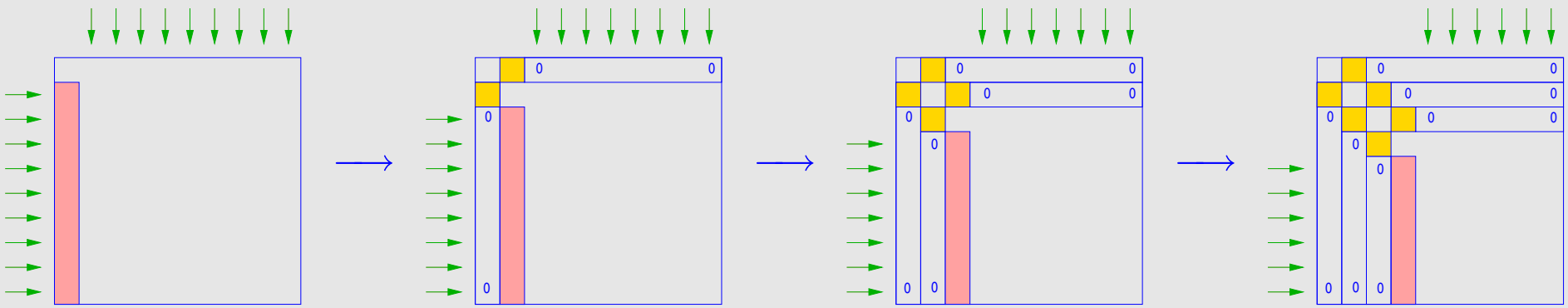
Computational cost: $O(n^3)$ operations per step of the QR-algorithm

Library implementations of the QR-algorithm provide *numerically stable* eigensolvers

△

Remark 4.2.3 (Unitary similarity transformation to tridiagonal form).

Successive Householder similarity transformations of $\mathbf{A} = \mathbf{A}^H$:
(→ $\hat{=}$ affected rows/columns, ▭ $\hat{=}$ targeted vector)



▶ transformation to tridiagonal form **!** (for general matrices a similar strategy can achieve a similarity transformation to upper Hessenberg form)

▶ this transformation is used as a preprocessing step for QR-algorithm ➢ `eig`.

Similar functionality for generalized EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$, $\mathbf{A}, \mathbf{B} \in \mathbb{C}^{n,n}$

`scipy.linalg.eig(a, b=None, left=False, right=True)`: also for generalized eigen value problems

Note: (Generalized) eigenvectors can be recovered as columns of $\mathbf{V}$:

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D} \quad \Leftrightarrow \quad \mathbf{A}(\mathbf{V})_{:,i} = (\mathbf{D})_{i,i}\mathbf{V}_{:,i} \ ,$$

if $\mathbf{D} = \mathrm{diag}(d_1, \dots, d_n)$.

Remark 4.2.4 (Computational effort for eigenvalue computations).

Computational effort (#elementary operations) for `eig()`:

eigenvalues & eigenvectors of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 25n^3 + O(n^2)$	
only eigenvalues of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 10n^3 + O(n^2)$	$O(n^3)$!
eigenvalues and eigenvectors $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 9n^3 + O(n^2)$	
only eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim \frac{4}{3}n^3 + O(n^2)$	
only eigenvalues of tridiagonal $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 30n^2 + O(n)$	

Sparse eigensolvers: ARPACK

scipy 0.7.1: `from scipy.sparse.linalg.eigen.arpack import arpack`
`arpack.eigen`, and `arpack.eigen_symmetric`

scipy 0.10: `scipy.sparse.linalg.eigs`

Note: `eig` not available in Matlab for sparse matrix arguments

Exception: `d=eig(A)` for sparse *Hermitian* matrices

△

Example 4.2.5 (Runtimes of `eig`).

Code 4.2.6: measuring runtimes of `eig`

```
import numpy as np
from scipy.sparse import lil_diags, lil_matrix
import scipy.sparse.linalg as sparsela
```

```
import time
from scipy.sparse.linalg.eigen.arpack import arpack

N = 500
A = np.random.rand(N,N); A = np.mat(A)
B = A.H*A
z = np.ones(N)

t0 = time.time()
C = lil_diags([3*z,z,z],[0,1,-1],(N,N))
t1 = time.time()
print 'C constructed in', t1-t0, 'seconds'
D = C.tocsr()

nexp = 4
ns = np.arange(5,N+1,5)
times = []
for n in ns:
    print 'n=', n
    An = A[:n,:n]; Bn = B[:n,:n]; Dn = D[:n,:n]

    ts = np.zeros(nexp)
    for k in xrange(nexp):
```

```
    ti = time.time()
    w = np.linalg.eigvals(An)
    tf = time.time()
    ts[k] = tf-ti
t1 = ts.sum()/nexp
print 't1 =', t1

ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w, V = np.linalg.eig(An)
    tf = time.time()
    ts[k] = tf-ti
t2 = ts.sum()/nexp
print 't2 =', t2

ts = np.zeros(nexp)
for k in xrange(nexp):
    ti = time.time()
    w = np.linalg.eigvalsh(Bn)
    tf = time.time()
    ts[k] = tf-ti
t3 = ts.sum()/nexp
```

```
    print 't3 =', t3

    ts = np.zeros(nexp)
    for k in xrange(nexp):
        ti = time.time()
        w, V = np.linalg.eigh(Bn)
        tf = time.time()
        ts[k] = tf-ti
    t4 = ts.sum()/nexp
    print 't4 =', t4

    ts = np.zeros(nexp)
    for k in xrange(nexp):
        ti = time.time()
        w, V = arpack.eigen_symmetric(Dn, k=n-1)
        tf = time.time()
        ts[k] = tf-ti
    t6 = ts.sum()/nexp
    print 't6 =', t6

    times += [np.array([t1, t2, t3, t4, t6])]
#
times = np.array(times)
```

```
from matplotlib import pyplot as plt
plt.loglog(ns, times[:,0], 'r+')
plt.loglog(ns, times[:,1], 'm*')
plt.loglog(ns, times[:,2], 'cp')
plt.loglog(ns, times[:,3], 'b^')
plt.loglog(ns, times[:,4])
plt.xlabel('matrix size n')
plt.ylabel('time [s]')
plt.title('eig runtimes')
plt.legend(('w = eig(A)', 'w, V = eig(A)', 'w = eig(B)', 'w, V = eig(B)',
    'w = arpack.eigen_symmetric(C)'), loc='upper left')
plt.savefig('eigtimingall.eps')
plt.show()

plt.clf()
plt.loglog(ns, times[:,0], 'r+')
plt.loglog(ns, times[:,1], 'm*')
plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,1], 'k-')
plt.xlabel('matrix size n')
plt.ylabel('time [s]')
plt.title('nXn random matrix')
plt.legend(('w = eig(A)', 'w, V = eig(A)', 'O(n^3)'), loc='upper left')
plt.savefig('eigtimingA.eps')
```

```
plt.show()

plt.clf()
plt.loglog(ns, times[:,2], 'cp')
plt.loglog(ns, times[:,3], 'b^')
plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,2], 'k-')
plt.xlabel('matrix size n')
plt.ylabel('time [s]')
plt.title('nXn random Hermitian matrix')
plt.legend(('w = eig(B)', 'w, V = eig(B)', 'O(n^3)'), loc='upper left')
plt.savefig('eigtimingB.eps')
plt.show()

plt.clf()
plt.loglog(ns, times[:,4], 'cp')
plt.loglog(ns, ns**2./(ns[0]**2.)*times[0,4], 'k-')
plt.xlabel('matrix size n')
plt.ylabel('time [s]')
plt.title('nXn random Hermitian matrix')
plt.legend(('w = arpack.eigen_symmetric(C)',  'O(n^2)'), loc='upper
  left')
plt.savefig('eigtimingC.eps')
plt.show()
```

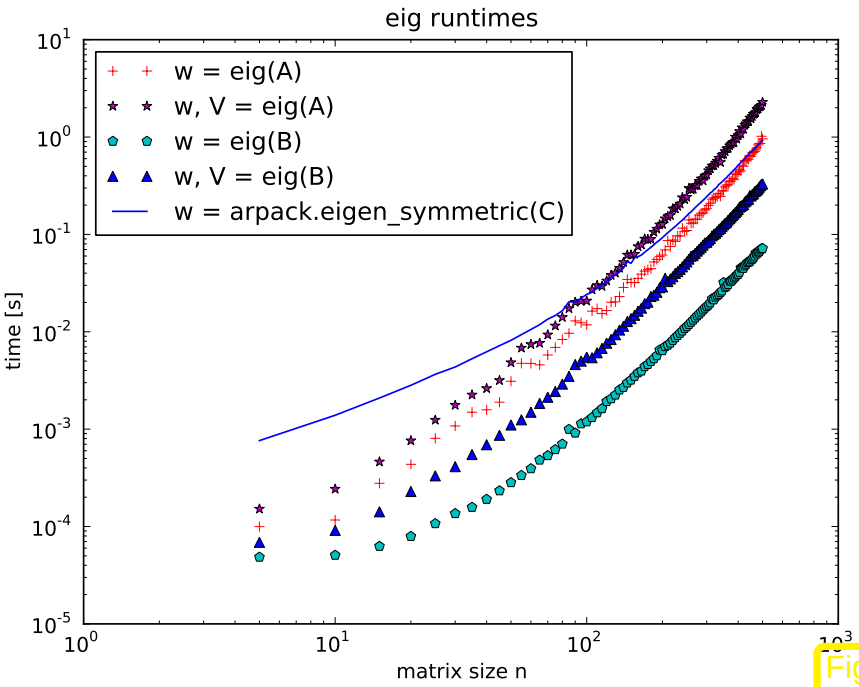

eig runtimes
w = eig(A)
w, V = eig(A)
w = eig(B)
w, V = eig(B)
w = arpack.eigen_symmetric(C)
time [s]
matrix size n

Fig. 34

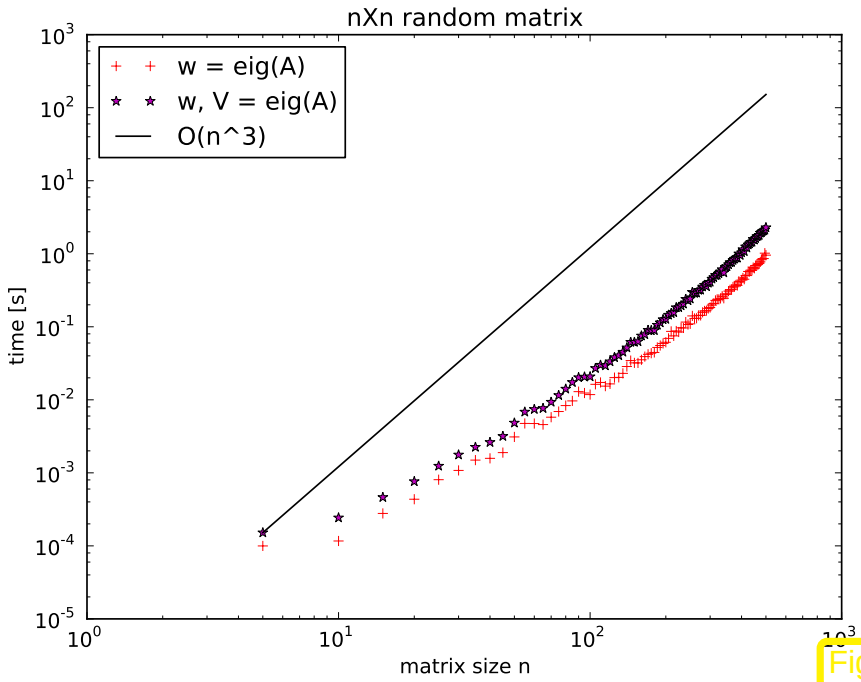

nXn random matrix
w = eig(A)
w, V = eig(A)
O(n^3)
time [s]
matrix size n

Fig. 35

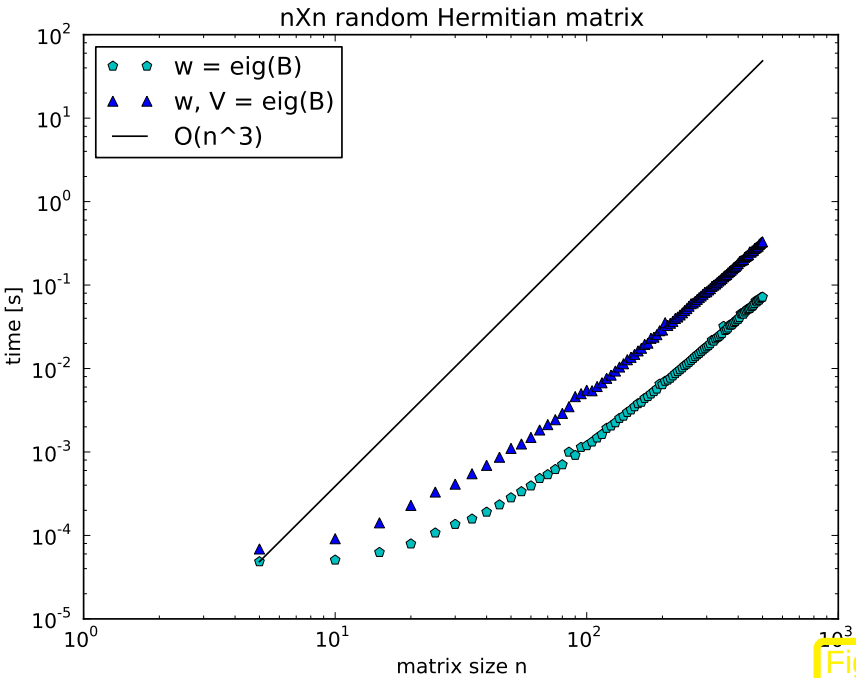


Fig. 36

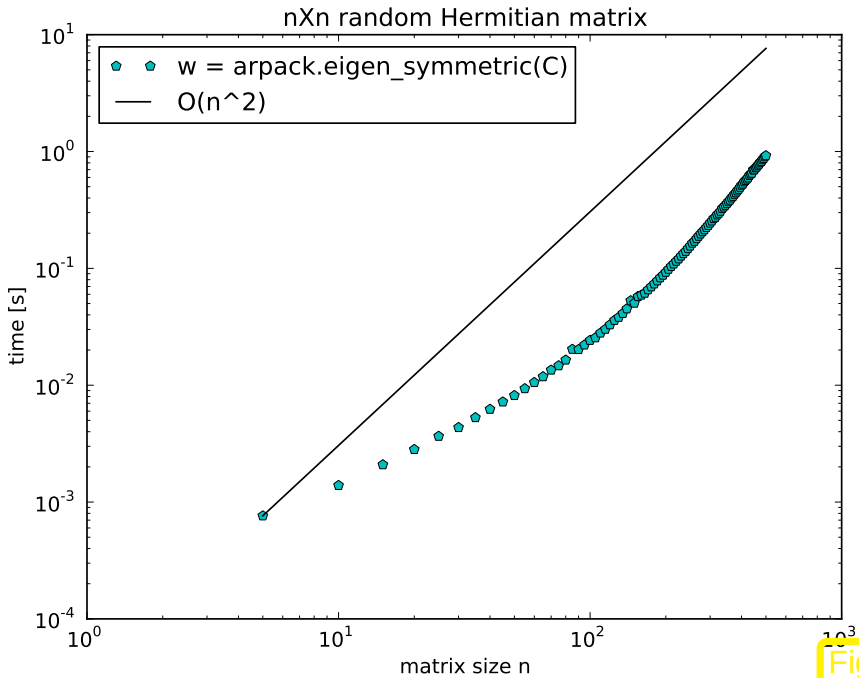


Fig. 37

☛ For the sake of efficiency: think which information you really need when computing eigenvalues/eigenvectors of dense matrices

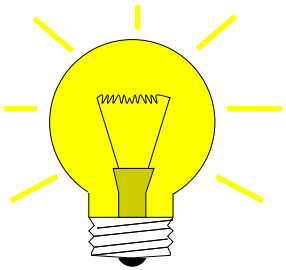
Potentially more efficient methods for *sparse matrices* will be introduced below in Sects. 4.3, 4.4.

◇

4.3 Power Methods

4.3.1 Direct power method

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find largest (in modulus) eigenvalue of $\mathbf{A}$ and (an) associated eigenvector.



Idea for $\mathbf{A} \in \mathbb{K}^{n,n}$ diagonalizable: $\mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$

$$\mathbf{z} = \sum_{j=1}^{n} \zeta_j (\mathbf{S})_{:,j} \quad \Rightarrow \quad \mathbf{A}^k \mathbf{z} = \sum_{j=1}^{n} \zeta_j \lambda_j^k \, (\mathbf{S})_{:,j} \; .$$

If $|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_{n-1}| < |\lambda_n|$, $\left\|(\mathbf{S})_{:,j}\right\|_2 = 1$, $j = 1, \dots, n$, $\zeta_n \neq 0$

$$\blacktriangleright \quad \frac{\mathbf{A}^k \mathbf{z}}{\left\|\mathbf{A}^k \mathbf{z}\right\|} \to \pm (\mathbf{S})_{:,n} \quad \text{= eigenvector for } \lambda_n \quad \text{for } k \to \infty \; . \tag{4.3.1}$$

Suggests direct power method (*ger.:* Potenzmethode): iterative method (→ Sect. 1.1)

$$\begin{aligned} &\text{initial guess:} \quad \mathbf{z}^{(0)} \text{ “arbitrary”} , \\ &\text{next iterate:} \quad \mathbf{w} := \mathbf{A}\mathbf{z}^{(k-1)} , \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2} , \quad k = 1, 2, \dots . \end{aligned} \tag{4.3.2}$$

Computational effort: $1\times$ matrix$\times$vector per step ➢ $\boxed{\text{inexpensive for sparse matrices}}$

$\mathbf{z}^{(k)} \longrightarrow$ eigenvector, but how do we get the associated eigenvalue λ_n **?**

❶ upon convergence from (4.3.1) ➢ $\mathbf{A}\mathbf{z}^{(k)} \approx \lambda_n \mathbf{z}^{(k)}$ ➢ $|\lambda_n| \approx \dfrac{\left\|\mathbf{A}\mathbf{z}^{(k)}\right\|}{\left\|\mathbf{z}^{(k)}\right\|}$

❷ for $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$: $\lambda_n \approx \operatorname*{argmin}_{\theta\in\mathbb{R}} \left\|\mathbf{A}\mathbf{z}^{(k)} - \theta\mathbf{z}^{(k)}\right\|_2^2$ ➤ $\lambda_n \approx \dfrac{(\mathbf{z}^{(k)})^T \mathbf{A}\mathbf{z}^{(k)}}{\left\|\mathbf{z}^{(k)}\right\|^2}$.

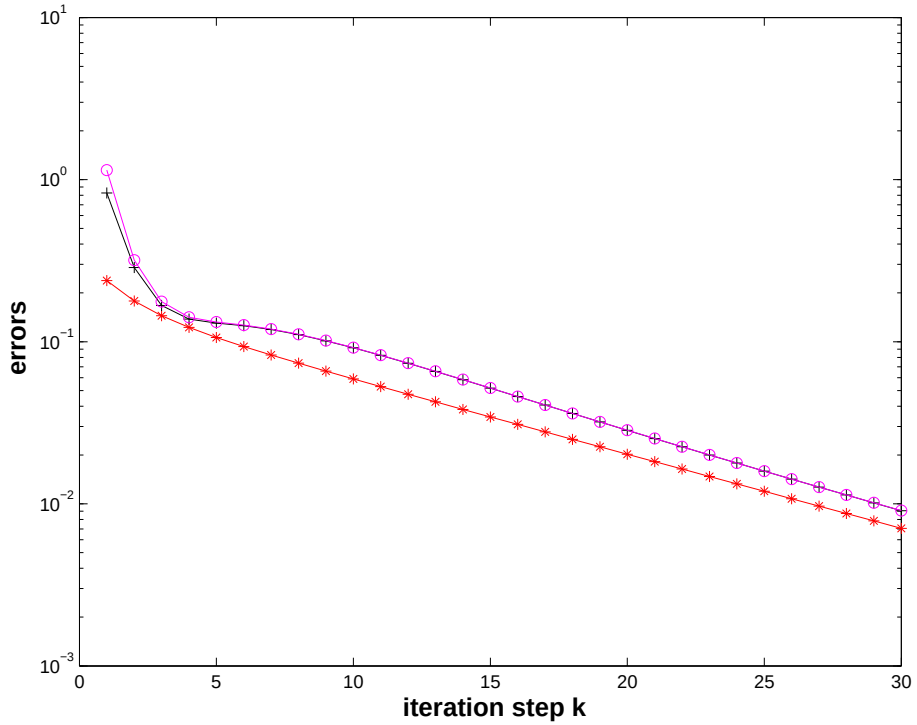
Definition 4.3.1. *For* $\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{u} \in \mathbb{K}^n$ *the Rayleigh quotient is defined by*

$$\rho_{\mathbf{A}}(\mathbf{u}) := \frac{\mathbf{u}^H \mathbf{A}\mathbf{u}}{\mathbf{u}^H \mathbf{u}} .$$

An immediate consequence of the definitions:

$$\lambda \in \sigma(\mathbf{A}) \quad , \quad \mathbf{z} \in \mathrm{Eig}_\lambda(\mathbf{A}) \quad \Rightarrow \quad \rho_{\mathbf{A}}(\mathbf{z}) = \lambda \; . \tag{4.3.3}$$

Example 4.3.1 (Direct power method).




```
n = len(d) # size of the matrix
S = triu(diag(r_[n:0:-1])+ones((n,
A = dot(S, dot(diag(d), inv(S)) )
```

◁

o : error $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

$*$: error norm $\left\| \mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n} \right\|$

$+$: $\left| \lambda_n - \dfrac{\left\| \mathbf{A}\mathbf{z}^{(k-1)} \right\|_2}{\left\| \mathbf{z}^{(k-1)} \right\|_2} \right|$

$\mathbf{z}^{(0)}$ = random vector

Test matrices:

① `d = 1.+r_[0:n]` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9$

② `d = ones(n); d[-1] = 2.` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.5$

③ `d = 1.-0.5**r_[1:5.1:0.5]` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9866$

$$\rho_{\mathrm{EV}}^{(k)} := \frac{\left\| \mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n} \right\|}{\left\| \mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot,n} \right\|} ,$$

$$\rho_{\mathrm{EW}}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|} .$$

	①		②		③	
k	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$
22	0.9102	0.9007	0.5000	0.5000	0.9900	0.9781
23	0.9092	0.9004	0.5000	0.5000	0.9900	0.9791
24	0.9083	0.9001	0.5000	0.5000	0.9901	0.9800
25	0.9075	0.9000	0.5000	0.5000	0.9901	0.9809
26	0.9068	0.8998	0.5000	0.5000	0.9901	0.9817
27	0.9061	0.8997	0.5000	0.5000	0.9901	0.9825
28	0.9055	0.8997	0.5000	0.5000	0.9901	0.9832
29	0.9049	0.8996	0.5000	0.5000	0.9901	0.9839
30	0.9045	0.8996	0.5000	0.5000	0.9901	0.9844

Observation: linear convergence (→ Def. 1.1.4)

◇

Theorem 4.3.2 (Convergence of direct power method)**.**
Let $\lambda_n > 0$ be the largest (in modulus) eigenvalue of $\mathbf{A} \in \mathbb{K}^{n,n}$ and have (algebraic) multiplicity 1. Let $\mathbf{v}, \mathbf{y}$ be the left and right eigenvectors of $\mathbf{A}$ for λ_n normalized according to $\|\mathbf{y}\|_2 = \|\mathbf{v}\|_2 = 1$. Then there is convergence

$$\left\|\mathbf{A}\mathbf{z}^{(k)}\right\|_2 \to \lambda_n \quad , \quad \mathbf{z}^{(k)} \to \pm\mathbf{v} \quad \textit{linearly with rate} \quad \frac{|\lambda_{n-1}|}{|\lambda_n|} \; ,$$

where $\mathbf{z}^{(k)}$ are the iterates of the direct power iteration and $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ is assumed.

Remark 4.3.2 (Initial guess for power iteration)*.*

roundoff errors ➤ $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ always satisfied in practical computations

Usual (not the best!) choice for $\mathbf{x}^{(0)}$ = random vector

△

Remark 4.3.3 (Termination criterion for direct power iteration)*.* (→ Sect. 1.1.2)

Adaptation of a posteriori termination criterion (1.2.7)

"relative change" $\leq$ `tol`:

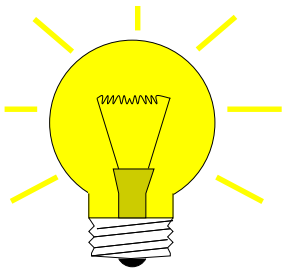
$$\begin{cases} \left\| \mathbf{z}^{(k)} - \mathbf{z}^{(k-1)} \right\| \leq (1/L - 1)\texttt{tol} \ , \\ \left| \dfrac{\left\| \mathbf{A}\mathbf{z}^{(k)} \right\|}{\left\| \mathbf{z}^{(k)} \right\|} - \dfrac{\left\| \mathbf{A}\mathbf{z}^{(k-1)} \right\|}{\left\| \mathbf{z}^{(k-1)} \right\|} \right| \leq (1/L - 1)\texttt{tol} \quad \text{see (1.1.17)} \ . \end{cases}$$

Estimated rate of convergence

△

4.3.2 Inverse Iteration

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find smallest (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.



If $\mathbf{A} \in \mathbb{K}^{n,n}$ regular:

$$\text{Smallest (in modulus) EV of } \mathbf{A} \quad = \quad \left(\text{Largest (in modulus) EV of } \mathbf{A}^{-1} \right)^{-1}$$

Direct power method (→ Sect. 4.3.1) for $\mathbf{A}^{-1}$ = inverse iteration

Code 4.3.4: inverse iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

```
import numpy as np
import scipy.linalg as splalg

def invit(A,tol):
    LUP = splalg.lu_factor(A, overwrite_a=True)
    n = A.shape[0]
    x = np.random.rand(n)
    x /= np.linalg.norm(x)
    splalg.lu_solve(LUP, x, overwrite_b=True)
    lold = 0
    lmin = 1./np.linalg.norm(x)
    x *= lmin
    while(abs(lmin-lold) > tol*lmin):
        lold = lmin
        splalg.lu_solve(LUP, x, overwrite_b=True)
        lmin = 1./np.linalg.norm(x)
        x *= lmin
    return lmin
```

Note: reuse of LU-factorization

Remark 4.3.5 (Shifted inverse iteration).

More general task:

For $\alpha \in \mathbb{C}$ find $\lambda \in \sigma(\mathbf{A})$ such that $|\alpha - \lambda| = \min\{|\alpha - \mu|, \mu \in \sigma(\mathbf{A})\}$

► Shifted inverse iteration:

$$\mathbf{z}^{(0)} \text{ arbitrary}, \quad \mathbf{w} = (\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{z}^{(k-1)} \quad , \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \dots , \tag{4.3.4}$$

where: $(\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{z}^{(k-1)} \hat{=}$ solve $(\mathbf{A} - \alpha\mathbf{I})\mathbf{w} = \mathbf{z}^{(k-1)}$ based on Gaussian elimination ($\leftrightarrow$ a single LU-factorization of $\mathbf{A} - \alpha\mathbf{I}$ as in Code 4.3.3).

What if "by accident" $\alpha \in \sigma(\mathbf{A})$ ($\Leftrightarrow$ $\mathbf{A} - \alpha\mathbf{I}$ singular) **?**

Stability of Gaussian elimination/LU-factorization will ensure that "$\mathbf{w}$ from (4.3.4) points in the right direction"

In other words, roundoff errors may badly affect the length of the solution $\mathbf{w}$, but not its direction.

Practice: If, in the course of Gaussian elimination/LU-factorization a zero pivot element is really encountered, then we just *replace it with* `eps`, in order to avoid `inf` values!

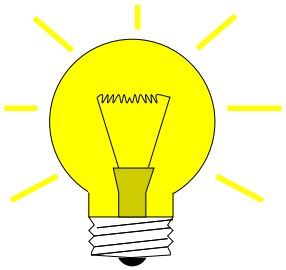
Thm. 4.3.2 ➢ Convergence of shifted inverse iteration for $\mathbf{A}^H = \mathbf{A}$:

Asymptotic linear convergence, Rayleigh quotient $\longrightarrow \lambda_j$ with rate

$$\frac{|\lambda_j - \alpha|}{\min\{|\lambda_i - \alpha|, i \neq j\}} \quad \text{with} \quad \lambda_j \in \sigma(\mathbf{A}) \,, \quad |\alpha - \lambda_j| \leq |\alpha - \lambda| \quad \forall \lambda \in \sigma(\mathbf{A}) \,.$$

▶ Extremely fast for $\alpha \approx \lambda_j$!

△



Idea: A posteriori adaptation of shift

Use $\alpha := \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})$ in k-th step of inverse iteration.

▼

Algorithm 4.3.6 (Rayleigh quotient iteration).

Rayleigh quotient iteration

(for normal $\mathbf{A} \in \mathbb{K}^{n,n}$)

Code 4.3.7: Rayleigh quotient iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

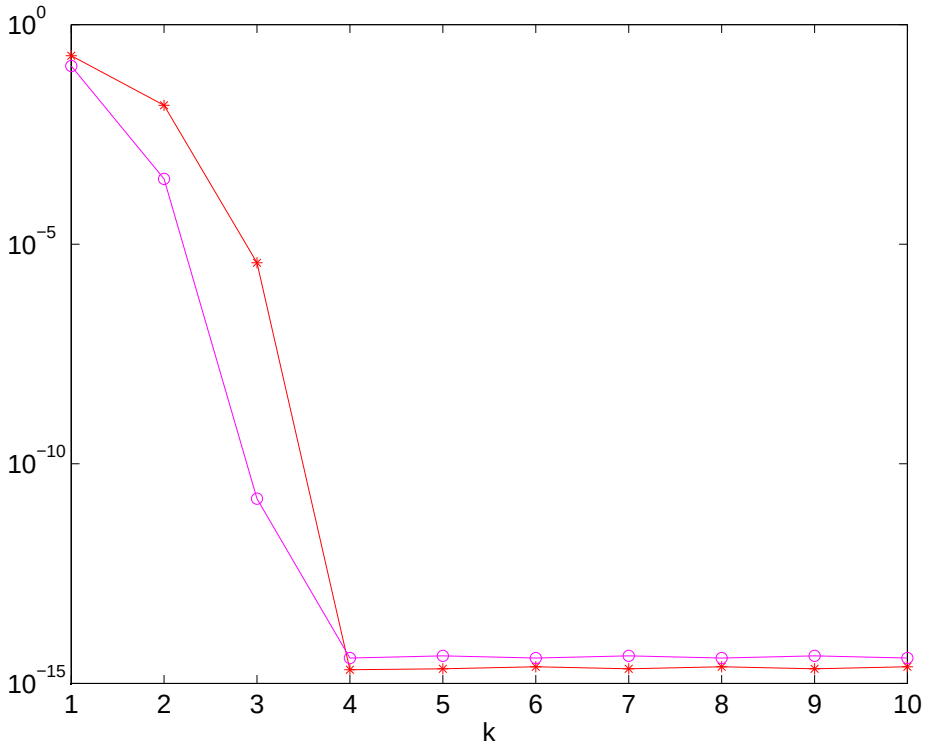
```
import numpy as np
def rqi(A,maxit):
    # For calculating the errors, the eigenvalues are calculated here with eig
    w, V = np.linalg.eig(A)
    t = np.where(w == abs(w).min())
    k = t[0];
    if len(k) > 1:
        print 'Error: no single smallest EV'
        raise ValueError
    ev = V[:,k[0]];  ev /= np.linalg.norm(ev)
    ew = w[k[0]]
    #
    n = A.shape[0]
    alpha = 0.
    z = np.random.rand(n); z /= np.linalg.norm(z)
    for k in xrange(maxit):
        z = np.linalg.solve(A-alpha*np.eye(n), z)
```

```
        z /= np.linalg.norm(z)
        alpha = np.dot(np.dot(A,z),z)
        ea = abs(alpha-ew)
        eb = np.min(np.linalg.norm(z-ev),np.linalg.norm(z+ev))
        print 'ea, eb =', ea, eb

def runit(d, tol, func):
    n = len(d) # size of the matrix
    Z = np.diag(np.sqrt(np.r_[n:0:-1])) + np.ones((n,n))
    Q,R = np.linalg.qr(Z)
    A = np.dot(Q, np.dot(np.diag(d), np.linalg.inv(Q)) )
    res = func(A,tol)
    print  res

# ------------------------------
if __name__=='__main__':
    n = 10 # size of the matrix
    d = 1.+np.r_[0:n] # eigenvalues
    print 'd =', d, 'd[-2]/d[-1] =', d[-2]/d[-1]
    runit(d, 10, rqi)
```

- Drawback compared with Code 4.3.3: reuse of LU-factorization no longer possible.
- Even if LSE nearly singular, stability of Gaussian elimination guarantees correct direction of $\mathbf{z}$, see discussion in Rem. 4.3.5.

Example 4.3.8 (Rayleigh quotient iteration).

Monitored: iterates of Rayleigh quotient iteration (4.3.6) for s.p.d. $\mathbf{A} \in \mathbb{R}^{n,n}$



$$\circ : \quad |\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$$

$$* : \quad \left\|\mathbf{z}^{(k)} - \mathbf{x}_j\right\|, \ \lambda_{\min} = \lambda_j, \ \mathbf{x}_j \in \mathrm{Eig}_{\mathbf{A}}(\lambda_j),$$

$$: \quad \left\|\mathbf{x}_j\right\|_2 = 1$$

k	$\lvert\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})\rvert$	$\left\lVert \mathbf{z}^{(k)} - \mathbf{x}_j \right\rVert$
1	0.09381702342056	0.20748822490698
2	0.00029035607981	0.01530829569530
3	0.00000000001783	0.00000411928759
4	0.00000000000000	0.00000000000000
5	0.00000000000000	0.00000000000000

Theorem 4.3.3. *If* $\mathbf{A} = \mathbf{A}^H$, *then* $\rho_{\mathbf{A}}(\mathbf{z}^{(k)})$ *converges* locally of order 3 *(→ Def. 1.1.13) to the smallest eigenvalue (in modulus), when* $\mathbf{z}^{(k)}$ *are generated by the Rayleigh quotient iteration* (4.3.6).

◇

4.3.3 Preconditioned inverse iteration (PINVIT)

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find smallest (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.

► Options: inverse iteration (→ Code 4.3.3) and Rayleigh quotient iteration (4.3.6).

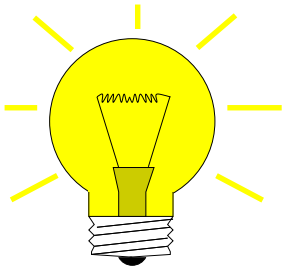
? What if direct solution of $\mathbf{Ax} = \mathbf{b}$ not feasible ?

This can happen, in case

- for large sparse $\mathbf{A}$ the amount of fill-in exhausts memory, despite sparse elimination techniques
- $\mathbf{A}$ is available only through a routine `evalA(x)` providing $\mathbf{A}\times$vector.

We expect that an approximate solution of the linear systems of equations encountered during inverse iteration should be sufficient, because we are dealing with approximate eigenvectors anyway.

Thus, iterative solvers for solving $\mathbf{A}\mathbf{w} = \mathbf{z}^{(k-1)}$ may be considered. However, the required accuracy is not clear a priori. Here we examine an approach that completely dispenses with an iterative solver and uses a *preconditioner* instead.



Idea: (for inverse iteration without shift, $\mathbf{A} = \mathbf{A}^H$ s.p.d.)

Instead of solving $\mathbf{A}\mathbf{w} = \mathbf{z}^{(k-1)}$ compute $\mathbf{w} = \mathbf{B}^{-1}\mathbf{z}^{(k-1)}$ with "inexpensive" s.p.d. approximate inverse $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$

$\rightarrow$ $\mathbf{B} \hat{=}$ Preconditioner for $\mathbf{A}$

Possible to replace $\mathbf{A}^{-1}$ with $\mathbf{B}^{-1}$ in inverse iteration **?**

NO, because we are not interested in smallest eigenvalue of $\mathbf{B}$!

Replacement $\mathbf{A}^{-1} \longrightarrow \mathbf{B}^{-1}$ possible only when applied to residual quantity

residual quantity **=** quantity that $\longrightarrow 0$ in the case of convergence to exact solution

Natural residual quantity for eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$:

$$\mathbf{r} := \mathbf{A}\mathbf{z} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z} \quad , \quad \rho_{\mathbf{A}}(\mathbf{z}) = \text{Rayleigh quotient} \longrightarrow \text{Def. 4.3.1 .}$$

Note: only *direction* of $\mathbf{A}^{-1}\mathbf{z}$ matters in inverse iteration (4.3.4)

$$(\mathbf{A}^{-1}\mathbf{z}) \parallel (\mathbf{z} - \mathbf{A}^{-1}(\mathbf{A}\mathbf{z} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z})) \quad \Rightarrow \quad \text{defines same next iterate!}$$

[Preconditioned inverse iteration (PINVIT) for s.p.d. $\mathbf{A}$]

$$\mathbf{z}^{(0)} \text{ arbitrary,} \quad \begin{aligned} \mathbf{w} &= \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{A}\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}) \,, \\ \mathbf{z}^{(k)} &= \frac{\mathbf{w}}{\|\mathbf{w}\|_2} \,, \end{aligned} \qquad k = 1, 2, \ldots \,. \tag{4.3.5}$$

Code 4.3.9: preconditioned inverse iteration (4.3.5)

```
from numpy import dot, ones, tile, array
from numpy.linalg import norm
from scipy.sparse import spdiags
from scipy.sparse.linalg.eigen.arpack import arpack
from scipy.sparse.linalg import spsolve

def pinvit(evA, invB, tol, maxit):
    n = A.shape[0]
    z = ones(n)/n
    res = []; rho = 0
    for k in xrange(maxit):
        v = evA(z)
        rhon = dot(v,z) # Rayleigh quotient
        r = v - rhon*z # residual
        z = z - invB(r) # iteration (4.3.5)
        z /= norm(z) # normalization
        res += [rhon] # tracking iteration
        if abs(rho-rhon) < tol*abs(rhon): break
```

```
        else: rho = rhon
    lmin = dot((evA(z)),z)
    res += [lmin]
    return lmin, z, res

def getdiag(A,k):
    # A in diagonal matrix format
    ind = abs(k+A.offsets).argmin()
    return A.data[ind,:]

# ------------------------------
if __name__=='__main__':
    k = 1; tol = 0; maxit = 50
    errC = []; errB = []
    import matplotlib.pyplot as plt
    for n in [20, 50, 100, 200]:
        a = tile([1./n, -1., 2*(1+1./n), -1., 1./n], (n,1))
        A = spdiags(a.T,[-n/2,-1,0,1,n/2],n,n)
        C = A.tocsr() # more efficient sparse format
        lam = min(abs(arpack.eigen_symmetric(C,n-1)[0]))
        evC = lambda x: C*x
        # inverse iteration
        invB = lambda x: spsolve(C,x)
        lmin, z, rn = pinvit(evC, invB, tol, maxit)
```

```
        err = abs(rn - lam)/lam
        errC += [err]
        txt = 'n='+str(n)
        plt.semilogy(err,'+', label='A^{-1}, '+ txt)
        print lmin, lam, err
        # preconditioned inverse iteration
        b = array([getdiag(A,-1), A.diagonal(), getdiag(A,1)])
        B = spdiags(b, [-1,0,1], n,n)
        B = B.tocsr()
        invB = lambda x: spsolve(B,x)
        lmin, z, rn = pinvit(evC, invB, tol, maxit)
        err = abs(rn - lam)/lam
        errB += [err]
        plt.semilogy(err,'o', label='B^{-1}, '+ txt)
        print lmin, lam, err

    plt.xlabel('# iteration step')
    plt.ylabel('error in approximation for λ_min')
    plt.legend()
    plt.savefig('pinvitD.eps')
    plt.show()
```

Computational effort:

1 matrix$\times$vector

1 evaluation of preconditioner

A few AXPY-operations

Example 4.3.10 (Convergence of PINVIT).

S.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, tridiagonal preconditioner

Monitored: error decay during iteration of Code 4.3.8: $|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_{\min}(\mathbf{A})|$

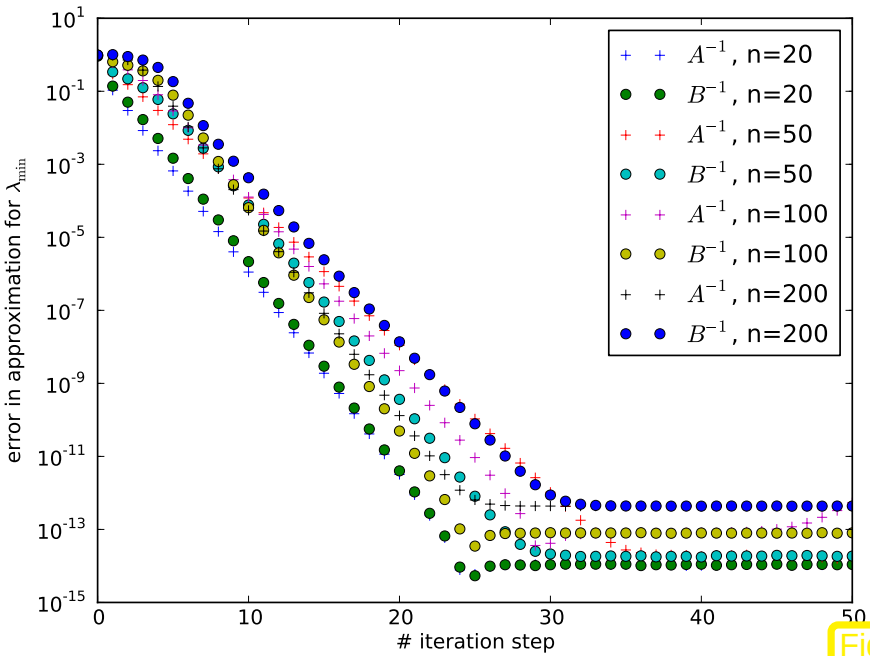


Fig. 38

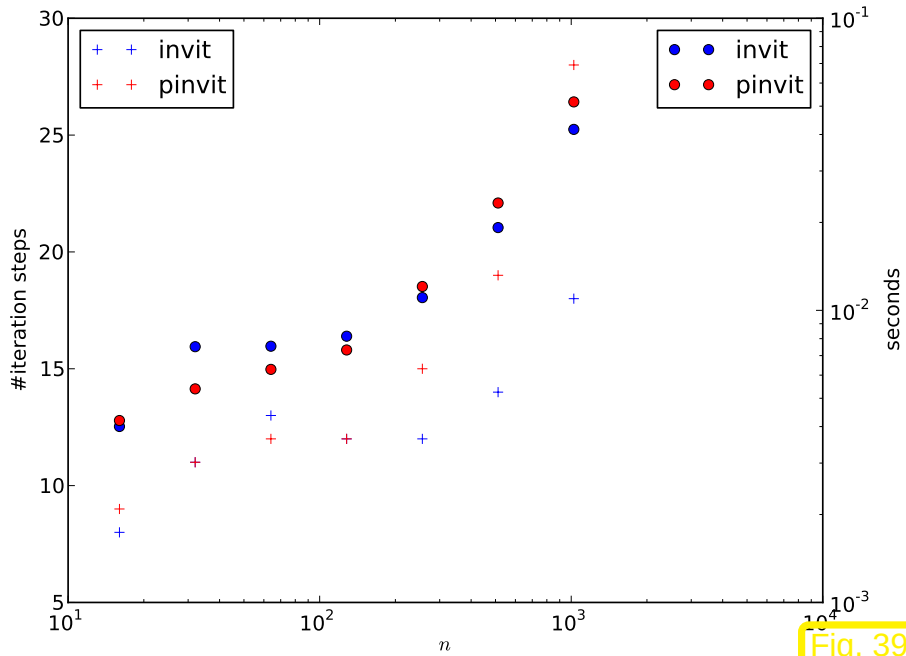


Fig. 39

Observation: linear convergence of eigenvectors also for PINVIT.

$\diamond$

Theory:
- linear convergence of (4.3.5)
- fast convergence, if spectral condition number $\kappa(\mathbf{B}^{-1}\mathbf{A})$ small

The theory of PINVIT is based on the identity

$$\mathbf{w} = \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)} + (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})(\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)}) \ . \tag{4.3.6}$$

For small residual $\mathbf{A}\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}$ PINVIT almost agrees with the regular inverse iteration.

4.3.4 Subspace iterations

Task:	Compute m, $m \ll n$, of the largest/smallest (in modulus) eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{C}^{n,n}$ and associated eigenvectors.

Recall that this task has to be tackled in step ❷ of the image segmentation algorithm Alg. **??**.

Preliminary considerations:

According to Cor. 4.1.7: For $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$ there is a factorization $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^T$ with $\mathbf{D} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$, $\lambda_j \in \mathbb{R}$, $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ $\mathbf{U}$ orthogonal. Thus, $\mathbf{u}_j := (\mathbf{U})_{:,j}$, $j = 1, \dots, n$, are (mutually orthogonal) eigenvectors of $\mathbf{A}$.

Assume $0 \leq \lambda_1 \leq \dots \leq \lambda_{n-2} < \lambda_{n-1} < \lambda_n$ (largest eigenvalues are simple).

If we just carry out the direct power iteration (4.3.2) for two vectors both sequencies will converge to the largest (in modulus) eigenvector. However, we recall that all eigenvectors are mutually orthogonal. This suggests that we orthogonalize the iterates of the second power iteration (that is to yield the eigenvector for the second largest eigenvalue) with respect to those of the first. This idea spawns the following iteration, *cf.* Gram-Schmidt orthogonalization in (**??**):

Code 4.3.11: one step of subspace power iteration, $m = 2$

```
v = A*v; w = A*w
v = v/norm(v); w = w - dot(v,w)*v; w = w/norm(w)
```

Analysis through eigenvector expansions ($\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $\|\mathbf{v}\|_2 = \|\mathbf{w}\|_2 = 1$)

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_j \mathbf{u}_j \quad , \quad \mathbf{w} = \sum_{i=1}^{n} \beta_j \mathbf{u}_j \ ,$$

$$\Rightarrow \quad \mathbf{A}\mathbf{v} = \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \quad , \quad \mathbf{A}\mathbf{w} = \sum_{i=1}^{n} \lambda_j \beta_j \mathbf{u}_j \ ,$$

$$\mathbf{v}_0 := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \Big(\sum_{i=1}^{n} \lambda_j^2 \alpha_j^2\Big)^{-1/2} \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \ ,$$

$$\mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w})\mathbf{v}_0 = \sum_{i=1}^{n} \left(\beta_j - \Big(\sum_{i=1}^{n} \lambda_j^2 \alpha_j \beta_j / \sum_{i=1}^{n} \lambda_j^2 \alpha_j^2 \Big) \alpha_j \right) \lambda_j \mathbf{u}_j \ .$$

We notice that $\mathbf{v}$ is just mapped to the next iterate in the regular direct power iteration (4.3.2). After many steps, it will be very close to $\mathbf{u}_n$, and, therefore, we may now assume $\mathbf{v} = \mathbf{u}_n \Leftrightarrow \alpha_j = \delta_{j,n}$ (Kronecker symbol).

$$\mathbf{z} := \mathbf{A}\mathbf{w} - (\mathbf{v}_0^T \mathbf{A}\mathbf{w})\mathbf{v}_0 = 0 \cdot \mathbf{u}_n + \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \ ,$$

$$\mathbf{w}^{(\text{new})} := \frac{\mathbf{z}}{\|\mathbf{z}\|_2} = \Big(\sum_{i=1}^{n-1} \lambda_j^2 \beta_j^2\Big)^{-1/2} \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \ .$$

The sequence $\mathbf{w}^{(k)}$ produced by repeated application of the mapping given by Code 4.3.10 asymptotically (that is, when $\mathbf{v}^{(k)}$ has already converged to $\mathbf{u}_n$) agrees with the sequence produced by the direct power method for $\widetilde{\mathbf{A}} := \mathbf{U}\,\mathrm{diag}(\lambda_1, \dots, \lambda_{n-1}, 0)$. Its convergence will be governed by the relative gap $\lambda_{n-1}/\lambda_{n-2}$, see Thm. 4.3.2.

Remark 4.3.12 (Generalized normalization).

The following two code snippets perform the same function, *cf.* Code 4.3.10: △

```
v = v/norm(v)
w = w - dot(v,w)*v; w = w/norm(w)
```

```
Q, R = qr(mat([v,w]))
v = Q[:,0]; w = Q[:,1]
```

Explanation ➢ Rem. 2.1.5

We revisit the above setting, Code 4.3.10. Is it possible to use the "$\mathbf{w}$-sequence" to accelerate the convergence of the "$\mathbf{v}$-sequence"?

Recall that by the min-max theorem Thm. **??**

$$\mathbf{u}_n = \operatorname{argmax}_{\mathbf{x}\in\mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{x}) \quad , \quad \mathbf{u}_{n-1} = \operatorname{argmax}_{\mathbf{x}\in\mathbb{R}^n,\, \mathbf{x}\perp\mathbf{u}_n} \rho_{\mathbf{A}}(\mathbf{x}) \; . \tag{4.3.7}$$

Idea: maximize Rayleigh quotient over $\operatorname{Span}\{\mathbf{v},\mathbf{w}\}$, where $\mathbf{v}$, $\mathbf{w}$ are output by Code 4.3.10. This leads to the optimization problem

$$(\alpha^*,\beta^*) := \operatorname*{argmax}_{\alpha,\beta\in\mathbb{R},\, \alpha^2+\beta^2=1} \rho_{\mathbf{A}}(\alpha\mathbf{v}+\beta\mathbf{w}) = \operatorname*{argmax}_{\alpha,\beta\in\mathbb{R},\, \alpha^2+\beta^2=1} \rho_{(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})}\left(\begin{pmatrix}\alpha\\ \beta\end{pmatrix}\right) \; . \tag{4.3.8}$$

Then a better approximation for the eigenvector to the largest eigenvalue is

$$\mathbf{v}^* := \alpha^*\mathbf{v} + \beta^*\mathbf{w} \; .$$

Note that $\|\mathbf{v}^*\|_2 = 1$, if both $\mathbf{v}$ and $\mathbf{w}$ are normalized, which is guaranteed in Code 4.3.10.

Then, orthogonalizing $\mathbf{w}$ w.r.t $\mathbf{v}^*$ will produce a new iterate $\mathbf{w}^*$.

Again the min-max theorem Thm. **??** tells us that we can find $(\alpha^*,\beta^*)^T$ as eigenvector to the largest eigenvalue of

$$(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \lambda\begin{pmatrix}\alpha\\ \beta\end{pmatrix} \; . \tag{4.3.9}$$

Since eigenvectors of symmetric matrices are mutually orthogonal, we find $\mathbf{w}^* = \alpha_2\mathbf{v} + \beta_2\mathbf{w}$, where $(\alpha_2,\beta_2)^T$ is the eigenvector of (4.3.9) belonging to the smallest eigenvalue. This assumes orthonormal vectors $\mathbf{v}$, $\mathbf{w}$.

Summing up :

Code 4.3.13: one step of subspace power iteration, $m = 2$, with Ritz projection

```
v = A*v; w = A*w; Q, R = qr(mat(v,w)); [U,D] = eig(Q.H*A*Q)
w = Q*U(:,1); v = Q*U(:,2)
```

General technique: Ritz projection

= "projection of a (symmetric) eigenvalue problem onto a subspace"

Example: Ritz projection of $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto $\mathrm{Span}\{\mathbf{v},\mathbf{w}\}$:

$$(\mathbf{v},\mathbf{w})^T\mathbf{A}(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \lambda(\mathbf{v},\mathbf{w})^T(\mathbf{v},\mathbf{w})\begin{pmatrix}\alpha\\ \beta\end{pmatrix} .$$

More general: Ritz projection of $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto $\mathrm{Im}(\mathbf{V})$ (subspace spanned by columns of $\mathbf{V}$)

$$\mathbf{V}^H\mathbf{A}\mathbf{V}\mathbf{w} = \lambda\mathbf{V}^H\mathbf{V}\mathbf{w} . \tag{4.3.10}$$

If $\mathbf{V}$ is unitary, then this generalized eigenvalue problem will become a standard linear eigenvalue problem.

Note that he orthogonalization step in Code 4.3.12 is actually redundant, if exact arithmetic could be employed, because the Ritz projection could also be realized by solving the generalized eigenvalue problem

However, prior orthogonalization is essential for numerical stability (→ Def. **??**), *cf.* the discussion in Sect. 2.1.

In implementations the vectors $\mathbf{v}$, $\mathbf{w}$ can be collected in a matrix $\mathbf{V} \in \mathbb{R}^{n,2}$:

Code 4.3.14: one step of subspace power iteration with Ritz projection, matrix version

```
V = A*V; Q,R = qr(V); U,D = eig(Q.H*A*Q); V = Q*U
```

Algorithm 4.3.15 (Subspace variant of direct power method with Ritz projection).

Assumption: $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$, $k \ll n$

Subspace variant of direct power method for s.p.d. $\mathbf{A}$

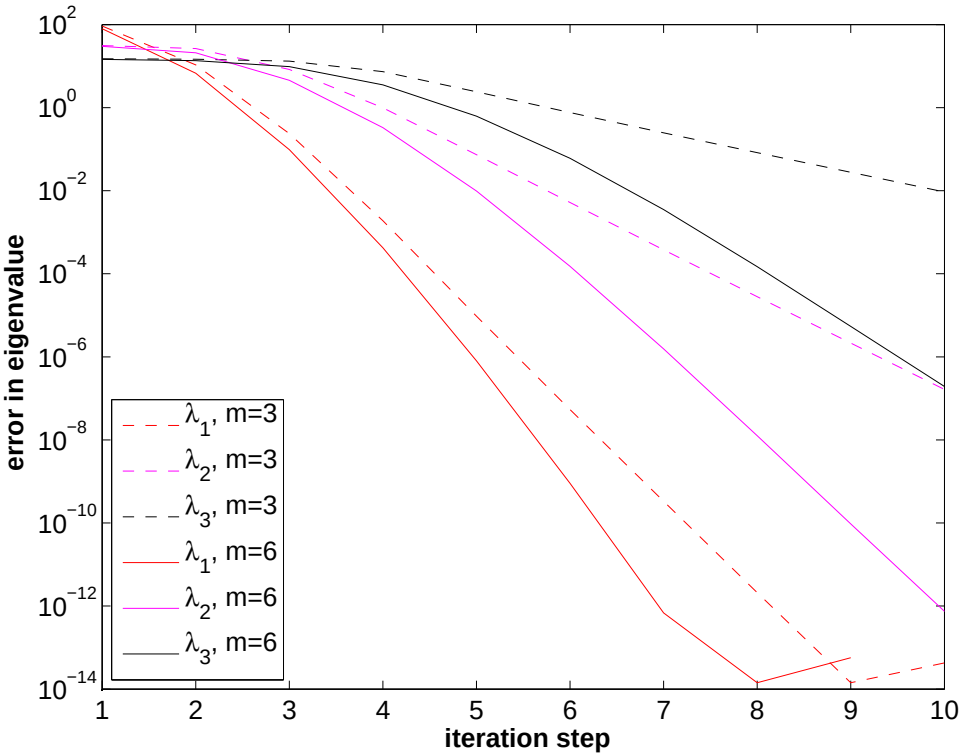
```
n = A.shape[0]; V = mat(rand(n,m)); d = mat(zeros()(k,1))
for i in xrange(maxit):
  V=A*V;
  Q, R = qr(V); Q = mat(Q)
  T=Q.T*A*Q;
  dn, S = eigh(T); S = mat(S)
  dn, S = sortEwEV(dn, S)
  V = Q*S
  res = norm(dn - d)
  if(res<tol):  break
  d = dn

return d[:k], V[:,:k]
```

(4.3.11)

Ritz projection

Generalized normalization to $\|\mathbf{z}\|_2 = 1$

Example 4.3.16 (Convergence of subspace variant of direct power method).



S.p.d. test matrix: $a_{ij} := \min\{\frac{i}{j}, \frac{j}{i}\}$

`n=200`

"Initial eigenvector guesses":

```
V = eye(n,m);
```

- Observation:
  linear convergence of eigenvalues
- choice $m > k$ boosts convergence of eigenvalues

◇

Remark 4.3.17 (Subspace power methods).

Analoguous to Alg. 4.3.15: construction of subspace variants of inverse iteration (→ Code 4.3.3), PINVIT (4.3.5), and Rayleigh quotient iteration (4.3.6).

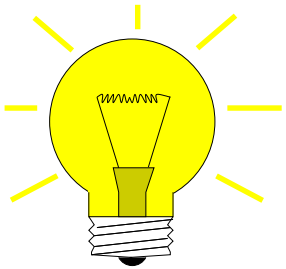
△

4.4 Krylov Subspace Methods

All power methods (→ Sect. 4.3) for the eigenvalue problem (EVP) $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ only rely on the last iterate to determine the next one (1-point methods, *cf.* (1.1.1))

➢ NO MEMORY, "Memory for power iterations": use information from previous iterates

➢ a direct method is fine for a small sparse matrix

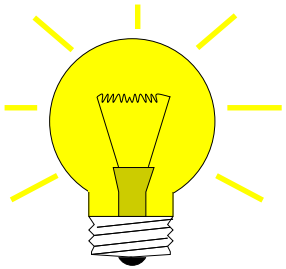


Idea: Better $\mathbf{v}^{(k)}$ from Ritz projection onto $V := \mathrm{Span}\left\{\mathbf{v}^{(0)}, \dots, \mathbf{v}^{(k)}\right\}$
(= space spanned by previous iterates)

Definition 4.4.1 (Krylov space)**.**
For $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{z} \in \mathbb{R}^n$, $\mathbf{z} \neq 0$, *the* l*-th Krylov space is defined as*

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) := \mathrm{Span}\left\{\mathbf{z}, \mathbf{A}\mathbf{z}, \dots, \mathbf{A}^{l-1}\mathbf{z}\right\} .$$

Equivalent definition: $\mathcal{K}_l(\mathbf{A},\mathbf{z}) = \{p(\mathbf{A})\mathbf{z}\colon p \text{ polynomial of degree } \leq l\}$



Idea:
- Orthonormal basis of $\mathcal{K}_l(\mathbf{A},\mathbf{z})$?
- use explicit Gram-Schmidt orthogonalization

Details: inductive approach: given $\{\mathbf{v}_1,\dots,\mathbf{v}_m\}$ ONB of $\mathcal{K}_m(\mathbf{A},\mathbf{z})$

$$\widetilde{\mathbf{v}}_{m+1} := \mathbf{A}\mathbf{v}_m - \sum_{j=1}^{m}(\mathbf{v}_j^H\mathbf{A}\mathbf{v}_m)\,\mathbf{v}_j\,, \quad \mathbf{v}_{m+1} := \frac{\widetilde{\mathbf{v}}_{m+1}}{\|\widetilde{\mathbf{v}}_{m+1}\|_2} \quad \Rightarrow \quad \mathbf{v}_{m+1} \perp \mathcal{K}_m(\mathbf{A},\mathbf{z})\,. \tag{4.4.1}$$

(Gram-Schmidt) orthogonal

Arnoldi process

In step l:

$1\times$	$\mathbf{A}\times$vector
$m+1$	dot products
m	AXPY-operations
n	divisions

➢ Computational cost for m steps, if at most k non-zero entries in each row of $\mathbf{A}$: $O(nkm^2)$

`H(m+1,m) = 0` → STOP !

Code 4.4.1: Arnoldi process

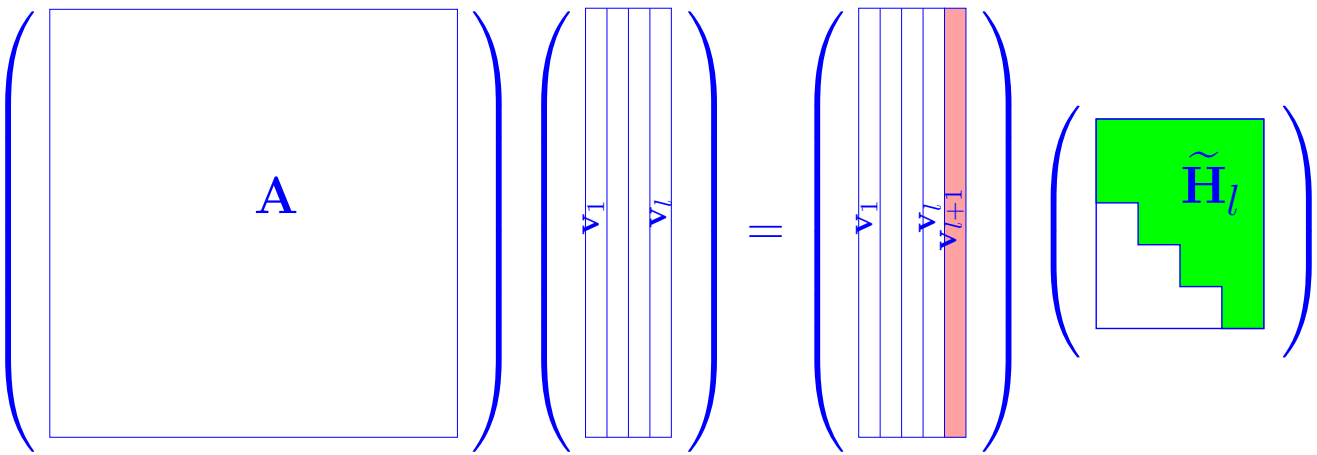
```
def arnoldi(A, v0, k):
    V = mat(zeros((v0.shape[0],k+1), dtype=complex))
    V[:,0] = v0.copy()/norm(v0)
    H = mat(zeros((k+1,k), dtype=complex))
    for m in xrange(k):
        vt = multMv(A, V[:, m])
        for j in xrange( m+1):
            H[ j, m] = (V[:, j].H * vt)[0,0]
            vt -= H[j, m] * V[:, j]
        H[m+1, m] = norm(vt);
        V[:,m+1] = vt.copy()/H[m+1, m]
    return V, H
```

If it does not stop prematurely, the Arnoldi process of Code 4.4.0 will yield an *orthonormal basis* (OBN) of $\mathcal{K}_{k+1}(\mathbf{A},\mathbf{v}_0)$ for a **general** $\mathbf{A}\in\mathbb{C}^{n,n}$.

Algebraic view of the Arnoldi process of Code 4.4.0, meaning of output H:

$$\mathbf{V}_l = \left[\mathbf{v}_1, \dots, \mathbf{v}_l\right] \;:\; \mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l \quad , \quad \widetilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l} \text{ mit } \widetilde{h}_{ij} = \begin{cases} \mathbf{v}_i^H \mathbf{A}\mathbf{v}_j & \text{, if } i \leq j \;, \\ \|\widetilde{\mathbf{v}}_i\|_2 & \text{, if } i = j+1 \;, \\ 0 & \text{else.} \end{cases}$$

➡ $\widetilde{\mathbf{H}}_l$ = non-square upper Hessenberg matrices



Translate Code 4.4.0 to matrix calculus:

Lemma 4.4.2 (Theory of Arnoldi process)**.**
For the matrices $\mathbf{V}_l \in \mathbb{K}^{n,l}$, $\widetilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l}$ *arising in the* l*-th step,* $l \leq n$*, of the Arnoldi process holds*

(i) $\mathbf{V}_l^H \mathbf{V}_l = \mathbf{I}$ *(unitary matrix),*

(ii) $\mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l$, $\widetilde{\mathbf{H}}_l$ *is non-square upper Hessenberg matrix,*

(iii) $\mathbf{V}_l^H \mathbf{A}\mathbf{V}_l = \mathbf{H}_l \in \mathbb{K}^{l,l}$, $h_{ij} = \widetilde{h}_{ij}$ *for* $1 \leq i,j \leq l$*,*

(iv) *If* $\mathbf{A} = \mathbf{A}^H$ *then* $\mathbf{H}_l$ *is tridiagonal (➢ Lanczos process)*

Proof. Direct from Gram-Schmidt orthogonalization and inspection of Code 4.4.0. □

Remark 4.4.2 (Arnoldi process and Ritz projection)*.*

Interpretation of Lemma 4.4.2 (iii) & (i):

$$\mathbf{H}_l\mathbf{x} = \lambda\mathbf{x} \text{ is a (generalized) Ritz projection of EVP } \mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

△

Eigenvalue approximation for general EVP $\mathbf{Ax} = \lambda\mathbf{x}$ by Arnoldi process:

In l-th step: $\lambda_n \approx \mu_l^{(l)}$, $\lambda_{n-1} \approx \mu_{l-1}^{(l)}, \dots, \lambda_1 \approx \mu_1^{(l)}$,

$$\sigma(\mathbf{H}_l) = \{\mu_1^{(l)}, \dots, \mu_l^{(l)}\}, \quad |\mu_1^{(l)}| \leq |\mu_2^{(l)}| \leq \dots \leq |\mu_l^{(l)}| \,.$$

Code 4.4.3: Arnoldi eigenvalue approximation

```
import scipy.linalg
import scipy.io
from scipy.linalg import norm
from scipy import mat, eye, zeros, array, arange, sqrt, real, imag
from arnoldi import multMv, arnoldi, randomvector
from scipy.sparse.linalg.eigen.arpack import speigs
from scipy.sparse.linalg.eigen.arpack import arpack
import time

def delta(i,j):
    if i==j: return 1
    else: return 0

def ConstructMatrix(N, case='minij'):
    H = mat(zeros([N,N]))
    for i in xrange(N):
        for j in xrange(N):
```

```
            if case=='sqrts':
                H[i,j] = 1L * sqrt((i+1)**2+(j+1)**2)+(i+1)*delta(i,j)
            else:
                H[i, j] = float( 1+min(i, j) )
    return H

def sortEwEV(d,v, focus=abs):
    u = v.copy()
    #I = abs(d).argsort()
    I = focus(d).argsort()
    a = d[I]
    u = u[:,I]
    return a,u

if __name__ == "__main__":
    # construct/import matrix
    #sparseformat = True
    sparseformat = False
    if sparseformat:
        # look at http://math.nist.gov/MatrixMarket/
        L = scipy.io.mmread('SomeMatrices/qc324.mtx.gz')
    else:
        #L = scipy.io.loadmat('Lmat.mat')['L']
        L = ConstructMatrix(1000)
```

```
    #L = ConstructMatrix(400,case='sqrts')
#print L
#————————————————————
N = L.shape[0]
nev = 10      # compute the largest nev eigenvalues and vectors
print 'full matrix approach'
t0 = time.clock()
if sparseformat:
    d0,V0 = scipy.linalg.eig(L.todense())
else:
    d0,V0 = scipy.linalg.eig(L)
print time.clock() - t0 , 'sec'
V0 = mat(V0)
# let us check if the eigenvectors are orthogonal
print abs(V0.H*V0 - eye(V0.shape[1])).max()
# consider only the largest in abs.value
a0,U0 = sortEwEV(d0,V0)
na = U0.shape[0]
a0 = a0[na-nev:]
U0 = U0[:,na-nev:]
print 'a0', a0
#————————————————————
print 'my arnoldi procedure'
# compute the largest na eigenvalues and vectors
```

```
# but only the first eigenvectors are relevant
v = randomvector(N)
na = 3*nev+1 # take 3 times more steps
if sparseformat:
    Lt = L.todense()
    t0 = time.clock()
    A,H = arnoldi(Lt, v, na)
else:
    t0 = time.clock()
    A,H = arnoldi(L, v, na)
print time.clock() - t0, 'sec'
d1,V1 =  scipy.linalg.eig(H[:-1,:])
Z = A[:,:-1]*V1
a1,U1 = sortEwEV(d1, Z)
a1 = a1[na-nev:]
U1 = U1[:,na-nev:]
print 'a1', a1
#-------------------
print 'same with ARPACK'
if sparseformat:
    L = L.tocsr()
    t0 = time.clock()
    d2,V2 = speigs.ARPACK_eigs(L.matvec,N,nev, which='LM')
else:
```

```
    t0 = time.clock()
    d2,V2 = arpack.eigen(L, k=nev, ncv=na, which='LM')
print time.clock() - t0 , 'sec'
a2,U2 = sortEwEV(d2,V2)
U2 = mat(U2)
print 'a2', a2
#-----------------------
e1 = abs(a0-a1)#/abs(a0)
e2 = abs(a0-a2)#/abs(a0)
print 'error in eigenvalues:\n' , e1, '\n', e2
print 'residuals in eigenvectors:'
z0 = [norm(L*U0[:,k] - a0[k]*U0[:,k]) for k in xrange(nev) ]
print 'eig(L)', z0
z1 = [norm(L*U1[:,k] - a1[k]*U1[:,k]) for k in xrange(nev) ]
print 'arnoldi(L)', z1
z2 = [norm(L*U2[:,k] - a2[k]*U2[:,k]) for k in xrange(nev) ]
print 'arpack(L)', z2
#-----------------------
# plot residuals in eigenvectors
from pylab import loglog, semilogy, plot, show, legend, xlabel,
  ylabel, savefig, rcParams#, xlim,
  ylim
plotf = loglog
#plotf = semilogy
```

```
params = {'backend': 'png',                    # or 'ps'
          'axes.labelsize': 20,
          'text.fontsize': 20,
          'legend.fontsize': 20,
          'xtick.labelsize': 16,
          'lines.markersize'  : 12,
          'ytick.labelsize': 16#,
          #'text.usetex': True
          }
rcParams.update(params)
#———————————————————
plotf(abs(a0), z0,'g-+')
plotf(abs(a0), z1,'r-o')
plotf(abs(a0), z2,'b-v')
legend(('eig', 'arnoldi', 'arpack'),2)
xlabel('|Eigenvalue|')
ylabel('Residual of Eigenvector')
savefig('resEVarpack.eps')
show()
#———————————————————
plotf(abs(a0),e1,'ro')
plotf(abs(a0),e2,'bv')
legend(('arnoldi', 'arpack'),2)
```

```
xlabel('|Eigenvalue|')
ylabel('Error')
savefig('errEWarpack.eps')
show()
```


Arnoldi process for computing the k largest (in modulus) eigenvalues of $\mathbf{A} \in \mathbb{C}^{n,n}$

1 $\mathbf{A}\times$vector per step

($\succ$ attractive for sparse matrices)

However: required storage increases with number of steps

If $\mathbf{A}^H = \mathbf{A}$, then $\mathbf{V}_m^T\mathbf{A}\mathbf{V}_m$ is a tridiagonal matrix.

$$\mathbf{V}_l^H \mathbf{A} \mathbf{V}_l = \begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & & \\ & & & & \ddots & \\ & & & \ddots & \ddots & \beta_{k-1} \\ & & & & \beta_{k-1} & \alpha_k \end{pmatrix} =: \mathbf{T}_l \in \mathbb{K}^{k,k} \quad \text{[tridiagonal matrix]}$$

Algorithm for computing $\mathbf{V}_l$ and $\mathbf{T}_l$:

Lanczos process

Computational effort/step:

$1\times$	$\mathbf{A}\times$vector
2	dot products
2	AXPY-operations
1	division

Code 4.4.4: Lanczos process

```
def lanczos(A, v0, k):
    V = mat(zeros((v0.shape[0],k+1)))
    alpha = zeros(k)
    bet = zeros(k+1)
    for m in xrange(k):
        vt = multMv(A , V[ :, m])
        if m > 0: vt -= bet[m] * V[:, m-1]
        alpha[m] = (V[:, m].H * vt)[0, 0]
        vt -= alpha[m] * V[:, m]
        bet[m+1] = norm(vt)
        V[:,m+1] = vt.copy()/bet[m+1]
        rbet = bet[1:-1]
        T = diag(alpha) + diag(rbet, 1) +
          diag(rbet, -1)
    return V, T
```

Total computational effort for l steps of Lanczos process, if $\mathbf{A}$ has at most k non-zero entries per row: $O(nkl)$

Note: Code 4.4.3 assumes that no residual vanishes. This could happen, if $\mathbf{z}_0$ exactly belonged to the span of a few eigenvectors. However, in practical computations inevitable round-off errors will always ensure that the iterates do not stay in an invariant subspace of $\mathbf{A}$, *cf.* Rem. 4.3.2.

Convergence (what we expect from the above considerations) $\longrightarrow$ [14, Sect. 8.5])

$$\text{In } l\text{-th step:} \quad \lambda_n \approx \mu_l^{(l)}\,,\ \lambda_{n-1} \approx \mu_{l-1}^{(l)}\,,\ldots,\ \lambda_1 \approx \mu_1^{(l)}\,,$$

$$\sigma(\mathbf{T}_l) = \{\mu_1^{(l)},\ldots,\mu_l^{(l)}\},\quad \mu_1^{(l)} \le \mu_2^{(l)} \le \cdots \le \mu_l^{(l)}\,.$$

Example 4.4.5 (Lanczos process for eigenvalue computation).

$\mathbf{A}$ with eigenvalues $1, 2, \ldots, 100$

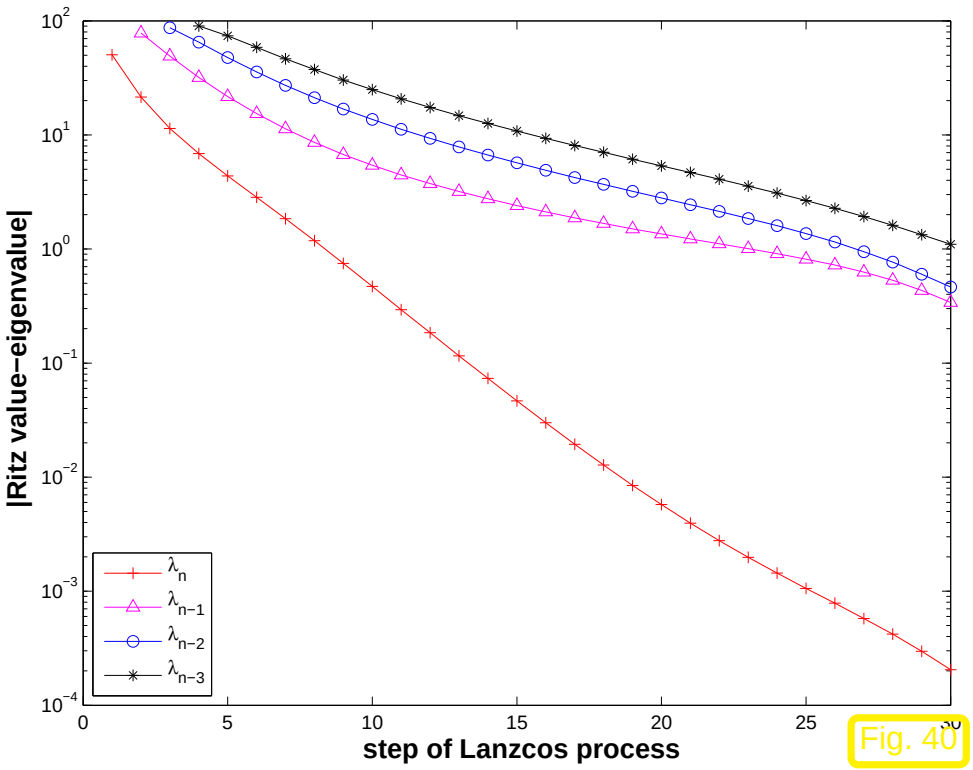


Fig. 40

`A = [min(i,j)], n=100`

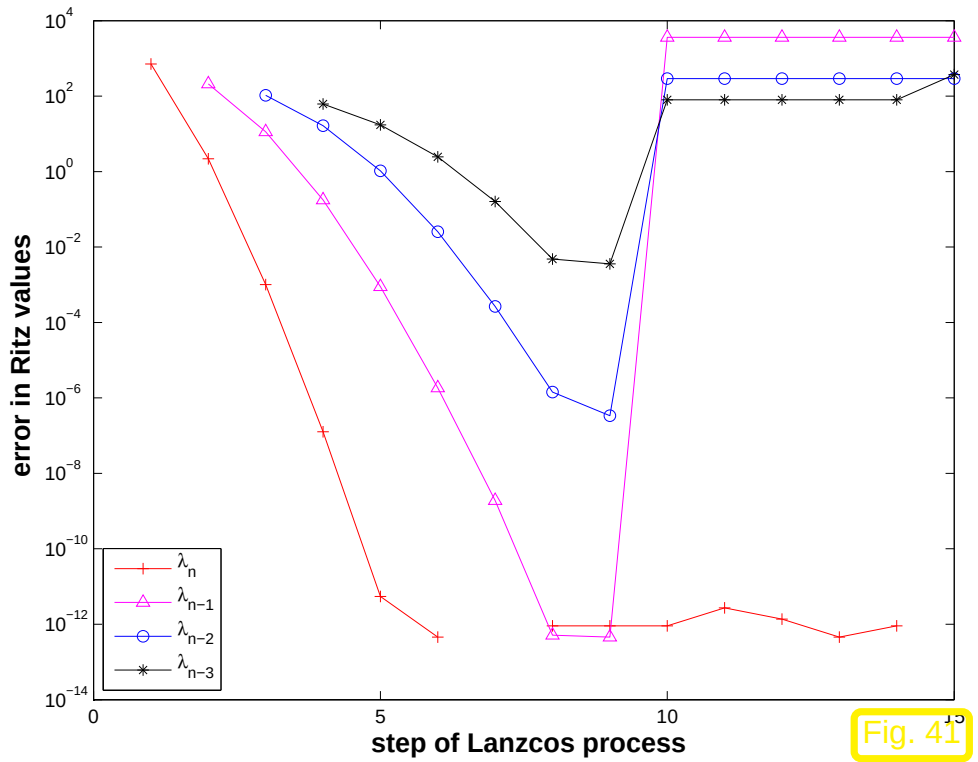


Fig. 41

Observation: linear convergence of Ritz values to eigenvalues.

However for $\mathbf{A} \in \mathbb{R}^{10,10}$, $a_{ij} = \min\{i,j\}$ good initial convergence, but sudden "jump" of Ritz values off eigenvalues!

Conjecture: Impact of roundoff errors

$\diamond$

Example 4.4.6 (Impact of roundoff on Lanczos process).

$$\mathbf{A} \in \mathbb{R}^{10,10} \quad , \quad a_{ij} = \min\{i,j\} \; .$$

```
A = [min(i,j)], n=10
```

Computed by `V, alpha, beta = lanczos(A,n,ones(n))`, see Code 4.4.3:

$$\mathbf{T} = \begin{pmatrix} 38.500000 & 14.813845 & & & & & & & & \\ 14.813845 & 9.642857 & 2.062955 & & & & & & & \\ & 2.062955 & 2.720779 & 0.776284 & & & & & & \\ & & 0.776284 & 1.336364 & 0.385013 & & & & & \\ & & & 0.385013 & 0.826316 & 0.215431 & & & & \\ & & & & 0.215431 & 0.582380 & 0.126781 & & & \\ & & & & & 0.126781 & 0.446860 & 0.074650 & & \\ & & & & & & 0.074650 & 0.363803 & 0.043121 & \\ & & & & & & & 0.043121 & 3.820888 & 11.991094 \\ & & & & & & & & 11.991094 & 41.254286 \end{pmatrix}$$

$$\sigma(\mathbf{A}) = \{0.255680, 0.273787, 0.307979, 0.366209, 0.465233, 0.643104, 1.000000, 1.873023, 5.048917, 44.766069\}$$

$$\sigma(\mathbf{T}) = \{0.263867, 0.303001, 0.365376, 0.465199, 0.643104, 1.000000, 1.873023, 5.048917, 44.765976, 44.766069\}$$

Uncanny cluster of computed eigenvalues of $\mathbf{T}$ ("ghost eigenvalues", [20, Sect. 9.2.5])

$$\mathbf{V}^H\mathbf{V} = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & 0.883711 \\ 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\ 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000328 \\ 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 \\ 0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\ 0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\ 0.883711 & 0.373799 & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Loss of orthogonality of residual vectors due to roundoff

(compare: impact of roundoff on CG iteration

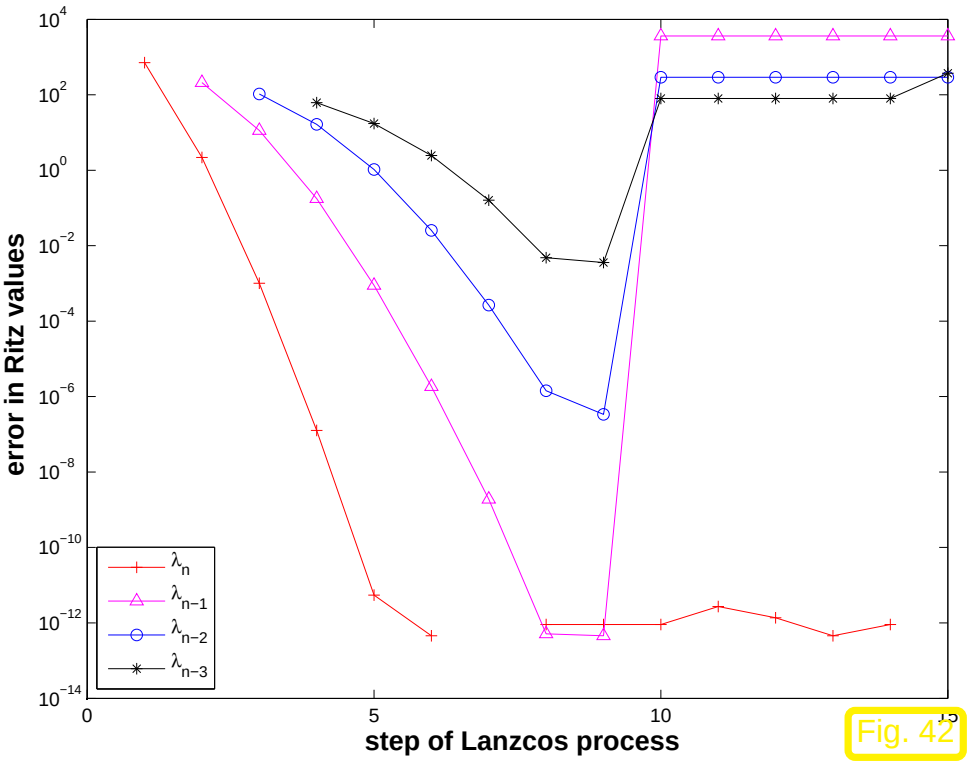
l	$\sigma(\mathbf{T}_l)$									
1										38.500000
2									3.392123	44.750734
3								1.117692	4.979881	44.766064
4							0.597664	1.788008	5.048259	44.766069
5						0.415715	0.925441	1.870175	5.048916	44.766069
6					0.336507	0.588906	0.995299	1.872997	5.048917	44.766069
7				0.297303	0.431779	0.638542	0.999922	1.873023	5.048917	44.766069
8			0.276160	0.349724	0.462449	0.643016	1.000000	1.873023	5.048917	44.766069
9		0.276035	0.349451	0.462320	0.643006	1.000000	1.873023	3.821426	5.048917	44.766069
10	0.263867	0.303001	0.365376	0.465199	0.643104	1.000000	1.873023	5.048917	44.765976	44.766069

$\diamond$

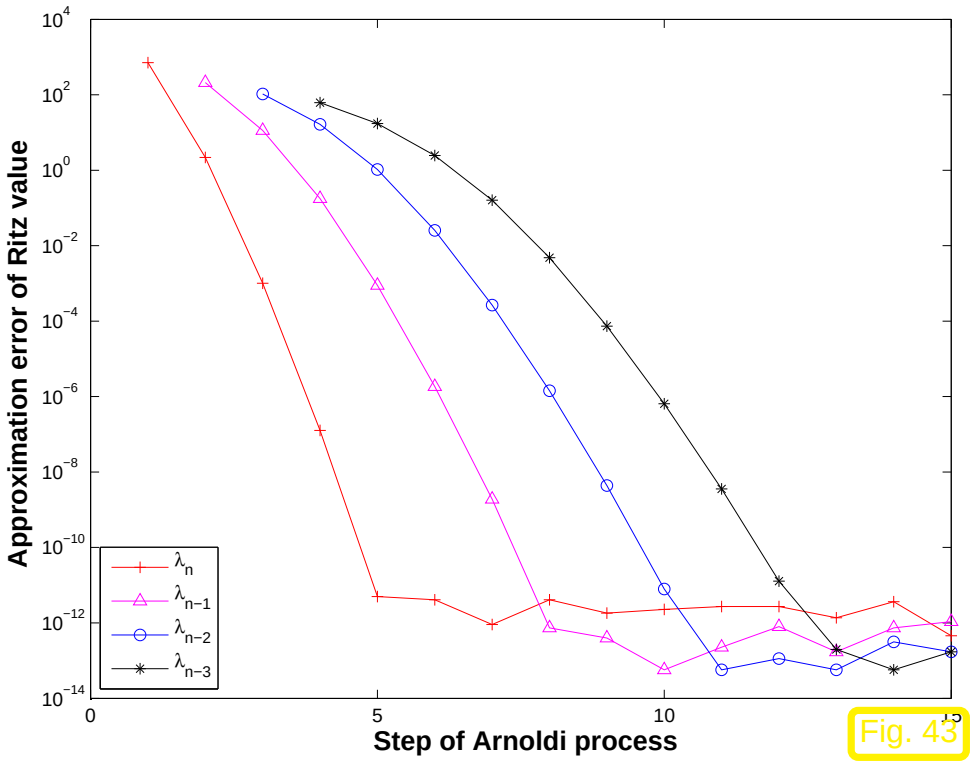
Example 4.4.7 (Stabilty of Arnoldi process).

$$\mathbf{A} \in \mathbb{R}^{100,100} \quad , \quad a_{ij} = \min\{i,j\} \; .$$

```
A = [min(i,j)], n=100
```



Lanczos process: Ritz values



Arnoldi process: Ritz values

Ritz values during Arnoldi process for `A = [min(i,j)], n =10);` $\leftrightarrow$ Ex. 4.4.5

l	$\sigma(\mathbf{H}_l)$									
1										38.500000
2									3.392123	44.750734
3								1.117692	4.979881	44.766064
4							0.597664	1.788008	5.048259	44.766069
5						0.415715	0.925441	1.870175	5.048916	44.766069
6					0.336507	0.588906	0.995299	1.872997	5.048917	44.766069
7				0.297303	0.431779	0.638542	0.999922	1.873023	5.048917	44.766069
8			0.276159	0.349722	0.462449	0.643016	1.000000	1.873023	5.048917	44.766069
9		0.263872	0.303009	0.365379	0.465199	0.643104	1.000000	1.873023	5.048917	44.766069
10	0.255680	0.273787	0.307979	0.366209	0.465233	0.643104	1.000000	1.873023	5.048917	44.766069

Observation: (almost perfect approximation of spectrum of $\mathbf{A}$)

For the above examples both the Arnoldi process and the Lanczos process are *algebraically equivalent*, because they are applied to a symmetric matrix $\mathbf{A} = \mathbf{A}^T$. However, they behave strikingly differently, which indicates that they are *not numerically equivalent*.

The Arnoldi process is much less affected by roundoff than the Lanczos process, because it does not take for granted orthogonality of the "residual vector sequence". Hence, the Arnoldi process enjoys superior numerical stability compared to the Lanczos process.

$\diamond$

Example 4.4.8 (Eigenvalue computation with Arnoldi process).

Eigenvalue approximation from Arnoldi process for *non-symmetric* $\mathbf{A}$, initial vector `ones(100);`

```
n=100
M = 2.*eye(n) + -0.5*eye(n,k=-1) + 1.5*eye(n, k=1); M = mat(M)
A = M*(diag(r_[1:n+1]))*M.I
```

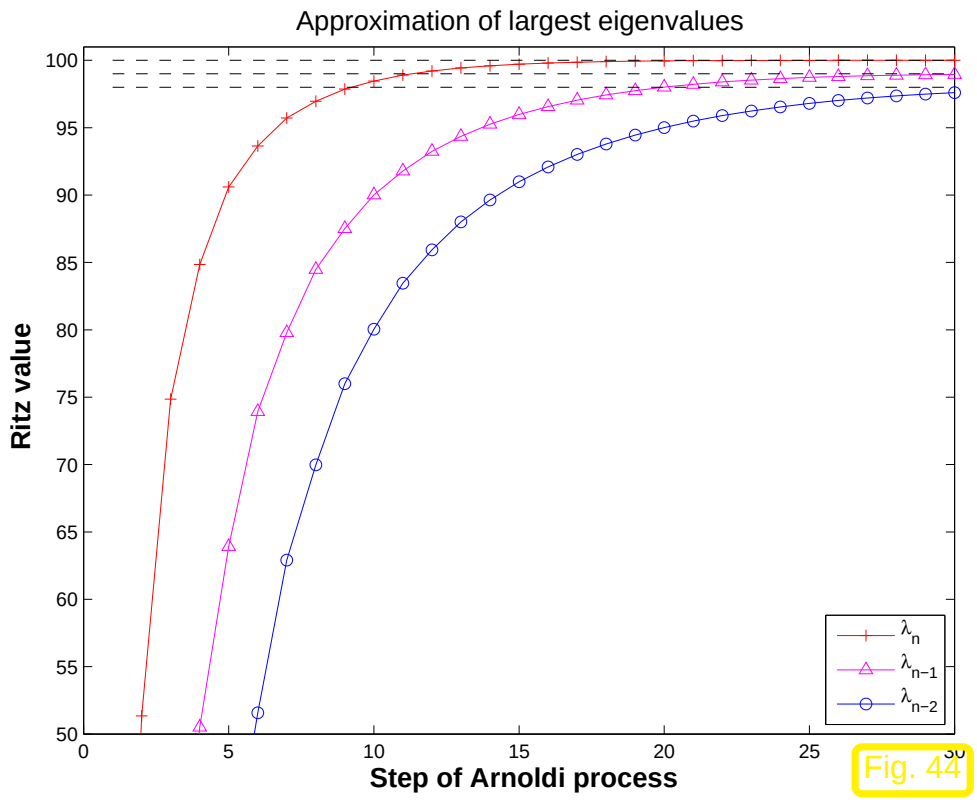


Fig. 44

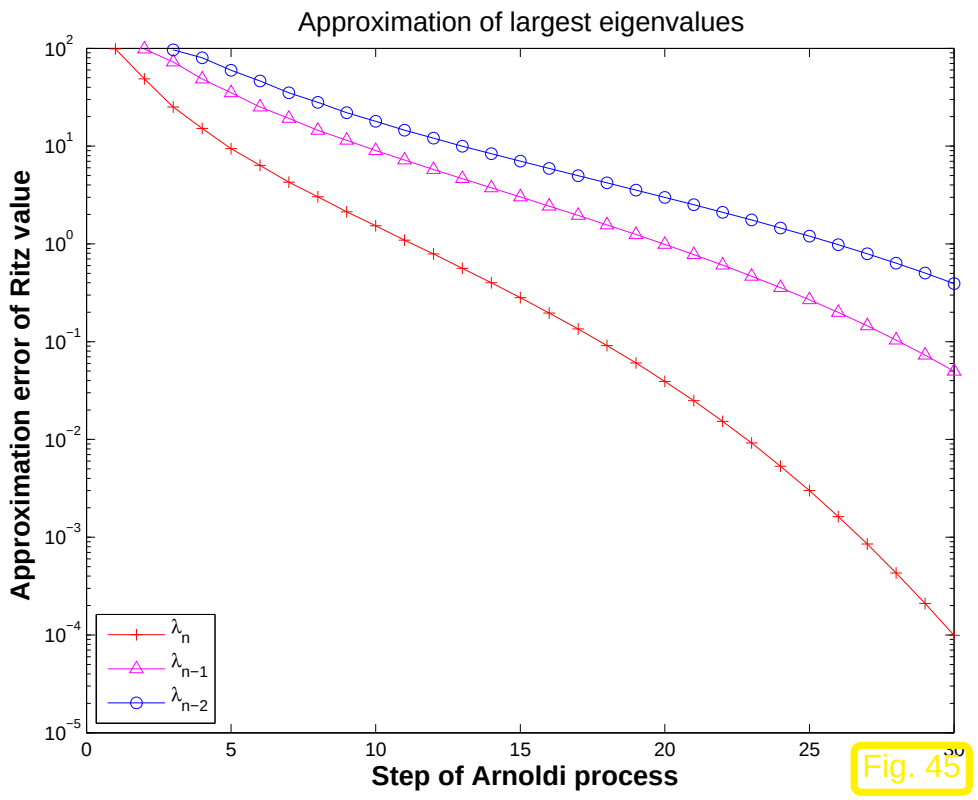


Fig. 45

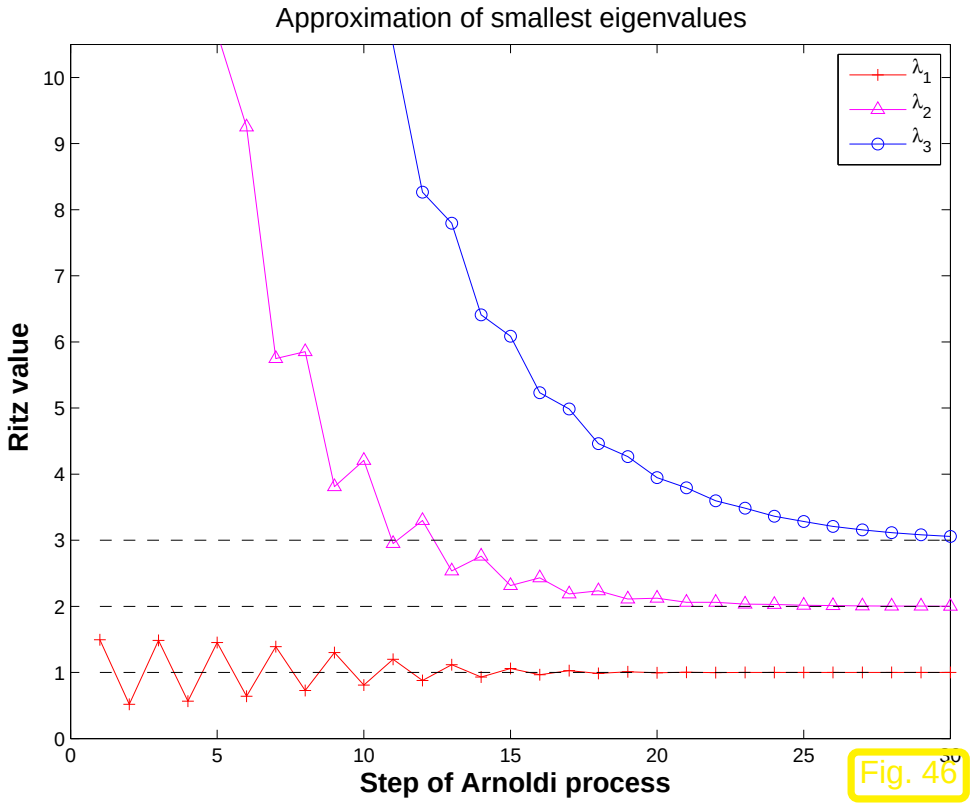


Fig. 46

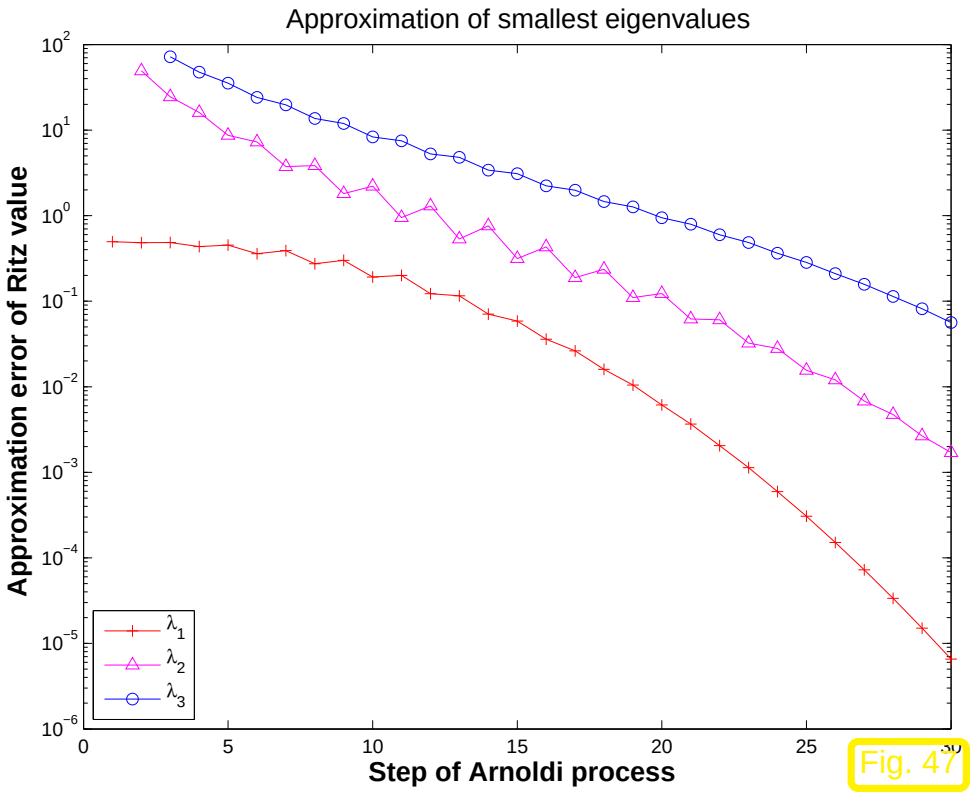


Fig. 47

Observation: "vaguely linear" convergence of largest and smallest eigenvalues

◇

Krylov subspace iteration methods (= Arnoldi process, Lanczos process) attractive for computing *a few* of the largest/smallest eigenvalues and associated eigenvectors of *large sparse matrices*.

Remark 4.4.9 (Krylov subspace methods for generalized EVP).

Adaptation of Krylov subspace iterative eigensolvers to generalized EVP: $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$, $\mathbf{B}$ s.p.d.: replace Euclidean inner product with "$\mathbf{B}$-inner product" $(\mathbf{x},\mathbf{y}) \mapsto \mathbf{x}^H\mathbf{B}\mathbf{y}$.

△

Python-functions:

`scipy.sparse.linalg.eigen.lobpcg`: solve sparse symmetric generalized eigenproblems with Locally Optimal Block Preconditioned Conjugate Gradient Method

`scipy.linalg.eig`: solve generalized eigenvalue problem of a square full matrix

MATLAB-functions:

`d = eigs(A,k,sigma)` : k largest/smallest eigenvalues of $\mathbf{A}$

`d = eigs(A,B,k,sigma)`: k largest/smallest eigenvalues for generalized EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$,$\mathbf{B}$ s.p.d.

`d = eigs(Afun,n,k)` : `Afun` = handle to function providing matrix×vector for $\mathbf{A}$/$\mathbf{A}^{-1}$/$\mathbf{A}-\alpha\mathbf{I}$/$(\mathbf{A}-\alpha\mathbf{B})^{-1}$. (Use flags to tell `eigs` about special properties of matrix behind `Afun`.)

`eigs` just calls routines of the open source ARPACK numerical library.

4.5 Essential Skills Learned in Chapter 4

You should know:

- complexity of the direct eigensolver eig of Matlab
- how the direct power method works and its convergence
- the idea behind the inverse power iteration, Rayleight quotient iteration and preconditioning
- what are Krylov methods, when and how to use them