05/09	Class 9	Dense direct solvers	Understand the principle of LU decomposition and the optimization and parallelization techniques that lead to the LINPACK benchmark.
05/12	Class 10	Dense eigensolvers	Determine eigenvalues and eigenvectors and understand the fast algorithms for diagonalization and orthonormalization.
05/16	Class 11	Sparse direct solvers	Understand reordering in AMD and nested dissection, and fast algorithms such as skyline and multifrontal methods.
05/19	Class 12	Sparse iterative solvers	Understand the notion of positive definiteness, condition number, and the difference between Jacobi, CG, and GMRES.
05/23	Class 13	Preconditioners	Understand how preconditioning affects the condition number and spectral radius, and how that affects the CG method.
05/26	Class 14	Multigrid methods	Understand the role of smoothers, restriction, and prolongation in the V-cycle.
05/30	Class 15	Fast multipole methods, H-matrices	Understand the concept of multipole expansion and low-rank approximation, and the role of the tree structure.

Dense linear algebra

Linear systems	$Ax = b$	$A = LDU$
Least squares	$\|\|Ax - b\|\|$	
Eigenvalues	$Ax = \lambda x$	$A = Q\Lambda Q^{-1}$
Singular values	$A^T Ax = \sigma^2 x$	$A = U\Sigma V$

numpy.linalg.eigvals

numpy.linalg.eigvals(*a*) [source]

Compute the eigenvalues of a general matrix.

Main difference between eigvals and eig: the eigenvectors aren't returned.

Parameters: **a** : *(..., M, M) array_like*

A complex- or real-valued matrix whose eigenvalues will be computed.

Returns: **w** : *(..., M,) ndarray*

The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

Raises: **LinAlgError**

If the eigenvalue computation does not converge.

See also:

eig	eigenvalues and right eigenvectors of general arrays
eigvalsh	eigenvalues of symmetric or Hermitian arrays.
eigh	eigenvalues and eigenvectors of symmetric/Hermitian arrays.

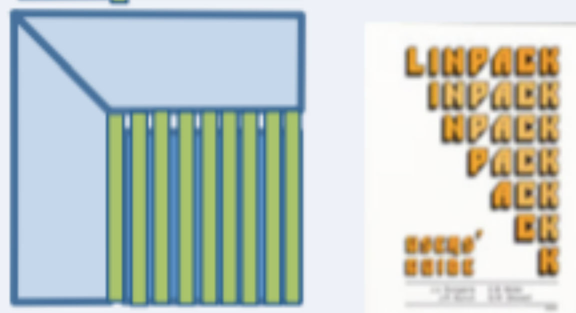
Notes

New in version 1.8.0.

Broadcasting rules apply, see the numpy.linalg documentation for details.

This is implemented using the _geev LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

LAPACK

Software/Algorithms follow hardware evolution in time		
EISPACK (70's) (Translation of Algol)		Rely on - Fortran, but row oriented
LINPACK (80's) (Vector operations)		Rely on - Level-1 BLAS operations - Column oriented
LAPACK (90's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (00's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (10's) New Algorithms (many-core friendly)		Rely on - DAG/scheduler - block data layout - some extra kernels

http://extremecomputingtraining.anl.gov/files/2015/08/dongarra-anl-0815.pdf

Eigenvalues & eigenvectors

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

$$A \in \mathbb{R}^{n \times n}$$

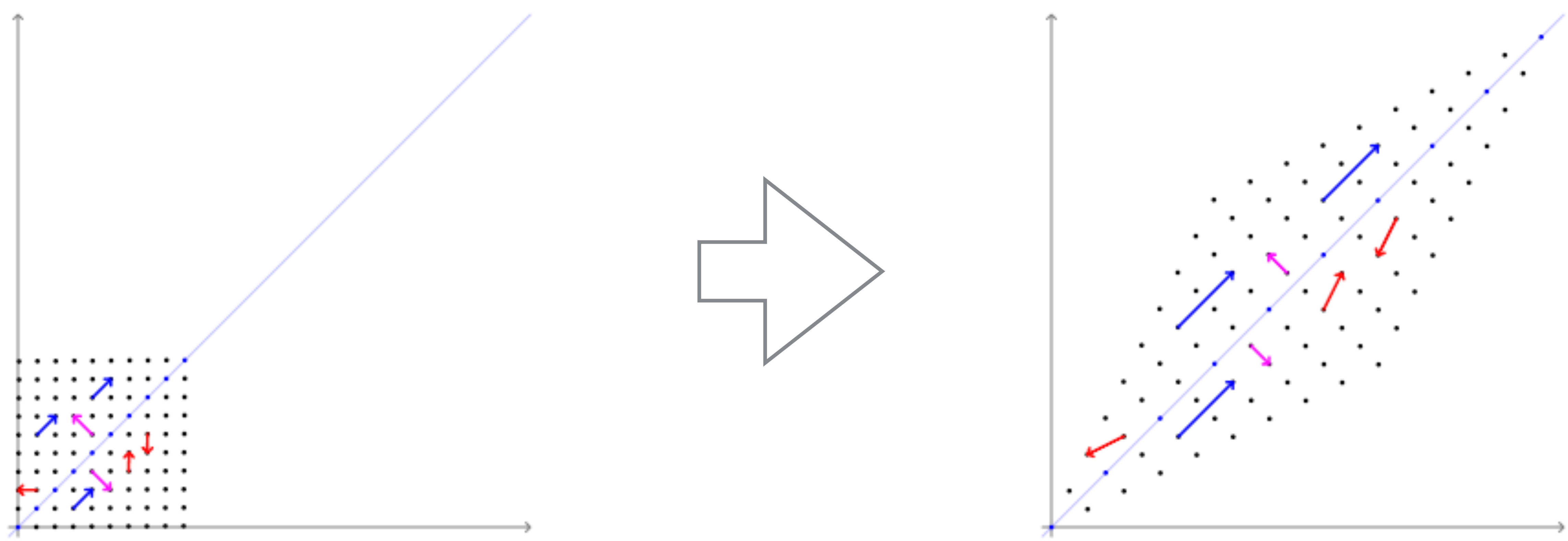
$$\lambda : \text{eigenvalue (scalar)}$$

$$\mathbf{x} : \text{eigenvector (vector)}$$

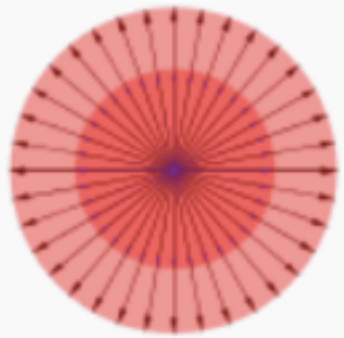
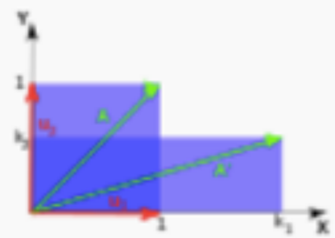
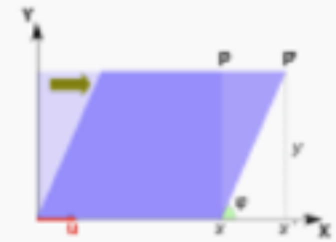
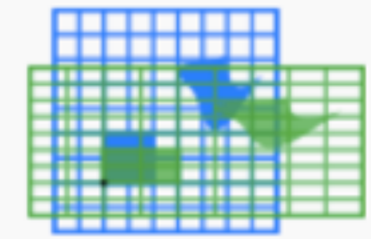
$$(\lambda, \mathbf{x}) : \text{eigenpair}$$

characteristic polynomial

$$|A - \lambda I| = 0$$



Eigenvalues of geometric transformations

	scaling	unequal scaling	rotation	horizontal shear	hyperbolic rotation
illustration					
matrix	$\begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix}$	$\begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix}$	$\begin{bmatrix} c & -s \\ s & c \end{bmatrix}$ $c = \cos\theta$ $s = \sin\theta$	$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} c & s \\ s & c \end{bmatrix}$ $c = \cosh\varphi$ $s = \sinh\varphi$
characteristic polynomial	$(\lambda - k)^2$	$(\lambda - k_1)(\lambda - k_2)$	$\lambda^2 - 2c\lambda + 1$	$(\lambda - 1)^2$	$\lambda^2 - 2c\lambda + 1$
eigenvalues λ_i	$\lambda_1 = \lambda_2 = k$	$\lambda_1 = k_1$ $\lambda_2 = k_2$	$\lambda_1 = e^{\mathbf{i}\theta} = c + s\mathbf{i}$ $\lambda_2 = e^{-\mathbf{i}\theta} = c - s\mathbf{i}$	$\lambda_1 = \lambda_2 = 1$	$\lambda_1 = e^{\varphi}$ $\lambda_2 = e^{-\varphi}$,
eigenvectors	All non-zero vectors	$u_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $u_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$u_1 = \begin{bmatrix} 1 \\ -\mathbf{i} \end{bmatrix}$ $u_2 = \begin{bmatrix} 1 \\ +\mathbf{i} \end{bmatrix}$	$u_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$u_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ $u_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

Eigenvalue algorithms

~~$|A - \lambda I| = 0$~~ polynomial root finding is an ill-conditioned problem

If A is Hermitian $A = Q\Lambda Q^*$ eigenvalue decomposition = singular value decomposition

If not … $A = QTQ^*$ T: upper-triangular

Schur factorization

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\text{phase 1}} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{\text{phase 2}} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}$$

$$A \neq A^* \qquad H \qquad T$$

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\text{phase 1}} \begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} \xrightarrow{\text{phase 2}} \begin{bmatrix} \times & & & & \\ & \times & & & \\ & & \times & & \\ & & & \times & \\ & & & & \times \end{bmatrix}$$

$$A = A^* \qquad T \qquad D$$

Householder transformation

$$
\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}
\xrightarrow{Q_1^T}
\underset{Q_1^T A}{\begin{bmatrix} \times & \times & \times & \times \\ \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \end{bmatrix}}
\xrightarrow{Q_1}
\underset{Q_1^T A Q_1}{\begin{bmatrix} \times & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \times & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \end{bmatrix}}
$$

$$
\underset{Q_1^T A Q}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ & \times & \times & \times \\ & \times & \times & \times \end{bmatrix}}
\xrightarrow{Q_2^T}
\underset{Q_2^T Q_1^T A Q_1}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ & 0 & \mathbf{x} & \mathbf{x} \end{bmatrix}}
\xrightarrow{Q_2}
\underset{Q_2^T Q_1^T A Q_1 Q_2}{\begin{bmatrix} \times & \times & \mathbf{x} & \mathbf{x} \\ \times & \times & \mathbf{x} & \mathbf{x} \\ & \times & \mathbf{x} & \mathbf{x} \\ & 0 & \mathbf{x} & \mathbf{x} \end{bmatrix}}
$$

$$
Q_{n-2}^T \cdots Q_2^T Q_1^T A Q_1 Q_2 \cdots Q_{n-2} = H = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ & \times & \times & \times \\ & & \times & \times \end{bmatrix}
$$

Fast eigenvalue algorithms

- Power iteration
- Inverse iteration
- Rayleigh quotient iteration
- Arnoldi iteration
- Lanczos algorithm
- QR algorithm

Power iteration

Determines one eigenvalue with largest absolute value

Useful when A is very large and sparse

Cannot find complex eigenvalues

$$
\begin{aligned}
&\text{Initialize} : q_0 = \text{a random vector} \\
&\text{for } k = 1, 2, \ldots \text{do} \\
&\quad z_k = A q_{k-1} \\
&\quad q_k = \frac{z_k}{||z_k||} \\
&\quad \lambda(k) = q_k^T A q_k \\
&\text{end for}
\end{aligned}
$$

Inverse iteration

$(A-\mu I)^{-1}$ has eigenpair $\left(\frac{1}{\lambda-\mu}, \mathbf{x}\right)$

Use a prior estimate of eigenvalue to get current eigenvalue

$$
\begin{aligned}
&\text{Initialize} : q_0 = \text{a random vector} \\
&\text{for } k = 1, 2, \ldots \text{do} \\
&\quad \text{Solve} : (A-\mu I) z_k = q_{k-1} \\
&\quad q_k = \frac{z_k}{||z_k||} \\
&\quad \lambda(k) = q_k^T A q_k \\
&\text{end for}
\end{aligned}
$$

Rayleigh quotient iteration

Replaces the estimated eigenvalue with the Rayleigh quotient

Faster convergence: quadratic in general and cubic for Hermitian matrix

$$\text{Initialize} : q_0 = \text{a random vector}$$

$$\text{for } k = 1, 2, \ldots \text{do}$$

$$\mu_{k-1} = \frac{q_{k-1}^T A q_{k-1}}{q_{k-1}^T q_{k-1}}$$

$$\text{Solve} : (A - \mu_{k-1} I) z_k = q_{k-1}$$

$$q_k = \frac{z_k}{||z_k||}$$

$$\lambda(k) = q_k^T A q_k$$

$$\text{end for}$$

Arnoldi iteration

Uses the stabilized Gram–Schmidt process to produce a sequence of orthonormal vectors

$$\text{Initialize}: q_0 = \text{a random vector } \textbf{with norm 1}$$

$$\text{for } k = 1, 2, \ldots \text{do}$$

$$q_k = A q_{k-1}$$

$$\text{for } j = 1, 2, \ldots \text{do}$$

$$h_{j,k-1} = q_j^* q_k$$

$$q_k = q_k - h_{j,k-1} q_j$$

$$\text{end for}$$

$$h_{k,k-1} = ||q_k||$$

$$q_k = \frac{q_k}{h_{k,k-1}}$$

$$\text{end for}$$

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

$$H = Q^* A Q$$

$$Q = \begin{bmatrix} \vdots \\ q_k \\ \vdots \end{bmatrix}$$

Lanczos algorithm

- Assume we have orthonormal vectors

$$\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_N$$

- Simply let $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k]$ hence

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$$

- We want to change $\mathbf{A}$ to a tridiagonal matrix $\mathbf{T}$, and apply a *similarly transformation*:

$$\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \mathbf{T} \text{ or } \mathbf{A}\mathbf{Q} = \mathbf{Q}\mathbf{T}$$

- So we define $\mathbf{T}$ to be

$$T_{k+1,k} = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \ldots & \ldots & \ldots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \ldots & \ldots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & 0 & \ldots & \vdots \\ \vdots & 0 & \ldots & \ldots & \ldots & \ldots & \vdots \\ \vdots & \ldots & \ldots & \ldots & \ldots & \ldots & \vdots \\ \vdots & \ldots & \ldots & \ldots & \ldots & \ldots & \beta_{k-1} \\ 0 & \ldots & \ldots & \ldots & 0 & \beta_{k-1} & \alpha_k \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 & \beta_k \end{bmatrix} \in \mathbb{C}^{k+1,k}$$

Lanczos algorithm

- After k steps we have $\mathbf{AQ}_k = \mathbf{Q}_{k+1}T_{k+1,k}$ for $\mathbf{A} \in \mathbb{C}^{N,N}$, $\mathbf{Q}_k \in \mathbb{C}^{N,k}$, $\mathbf{Q}_{k+1} \in \mathbb{C}^{N,k+1}$, $\mathbf{T}_{k+1,k} \in \mathbb{C}^{k+1,k}$.
- We observe that

$$\mathbf{AQ}_k = \mathbf{Q}_{k+1}T_{k+1,k} = \mathbf{Q}_k T_{k,k} + \beta_k \mathbf{q}_{k+1}\mathbf{e}_k^T$$

- Now $\mathbf{AQ} = \mathbf{QT}$ hence

$$A\,[\mathbf{q}_1,\, \mathbf{q}_2,\, \ldots,\, \mathbf{q}_k] = [\mathbf{q}_1,\, \mathbf{q}_2,\, \ldots,\, \mathbf{q}_k]\,T_k$$

- The first column of the left hand side matrix is given by

$$\mathbf{Aq}_1 = \alpha_1\mathbf{q}_1 + \beta_1\mathbf{q}_2$$

- The ith term by

$$\mathbf{Aq}_i = \beta_{i-1}\mathbf{q}_{i-1} + \alpha_i\mathbf{q}_i + \beta_i\mathbf{q}_{i+1},^{\dagger} \quad i = 2, \ldots$$

- We wish to find the alphas and betas so multiply † by $\mathbf{q}_i^T$ so that

$$\begin{aligned}
\mathbf{q}_i^T\mathbf{Aq}_i &= \mathbf{q}_i^T\beta_{i-1}\mathbf{q}_{i-1} + \mathbf{q}_i^T\alpha_i\mathbf{q}_i + \mathbf{q}_i^T\beta_i\mathbf{q}_{i+1} \\
&= \beta_{i-1}\mathbf{q}_i^T\mathbf{q}_{i-1} + \alpha_i\mathbf{q}_i^T\mathbf{q}_i + \beta_i\mathbf{q}_i^T\mathbf{q}_{i+1} \\
&= \alpha_i\mathbf{q}_i^T\mathbf{q}_i
\end{aligned}$$

- We obtain β_i by rearranging † from the recurrence formula

$$\mathbf{r}_i \equiv \beta_i\mathbf{q}_{i+1} = \mathbf{Aq}_i - \alpha_i\mathbf{q}_i - \beta_{i-1}\mathbf{q}_{i-1}$$

- We assume $\beta_i \neq 0$ and so $\beta_i = ||\mathbf{r}_i||_2$.

Lanczos algorithm

$$\text{Initialize} : q_0 = 0, q_1 = b/||b||, \beta_0 = 0$$

$$\text{for } k = 1, 2, ... \text{do}$$

$$v = Aq_k$$

$$\alpha_k = q_k^T v$$

$$v = v - \beta_{k-1} q_{k-1} - \alpha_k q_k$$

$$\beta_k = ||v||$$

$$q_{k+1} = v/\beta_k$$

$$\text{end for}$$

QR algorithm

QR factorization of A at step k ⟶ $A_k = Q_k R_k$

A at step k+1 $A_{k+1} = R_k Q_k$

$$\text{Initialize} : A_0 = A$$
$$\text{for } k = 1, 2, ...\text{do}$$
$$\quad Q_k R_k = A_{k-1}$$
$$\quad A_{k+1} = R_k Q_k$$
$$\text{end for}$$

Practical QR algorithm

1. Before starting the iteration, A is reduced to tridiagonal form
2. Instead of A_k a shifted matrix $A_k - \mu_k I$ is factored
3. Whenever an eigenvalue is found, the problem is deflated by breaking A_k into submatrices

$$\text{Initialize}: Q_0^T A_0 Q_0 = A \ (\text{tridiagonal } A_0)$$

for $k = 1, 2, \ldots$do

$$\mu_k = A_{k,mm}$$

$$Q_k R_k = A_{k-1} - \mu_k I$$

$$A_k = R_k Q_k + \mu_k I$$

If any off diagonal element $A_{j,j+1}$ is sufficiently close to 0,

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

end for

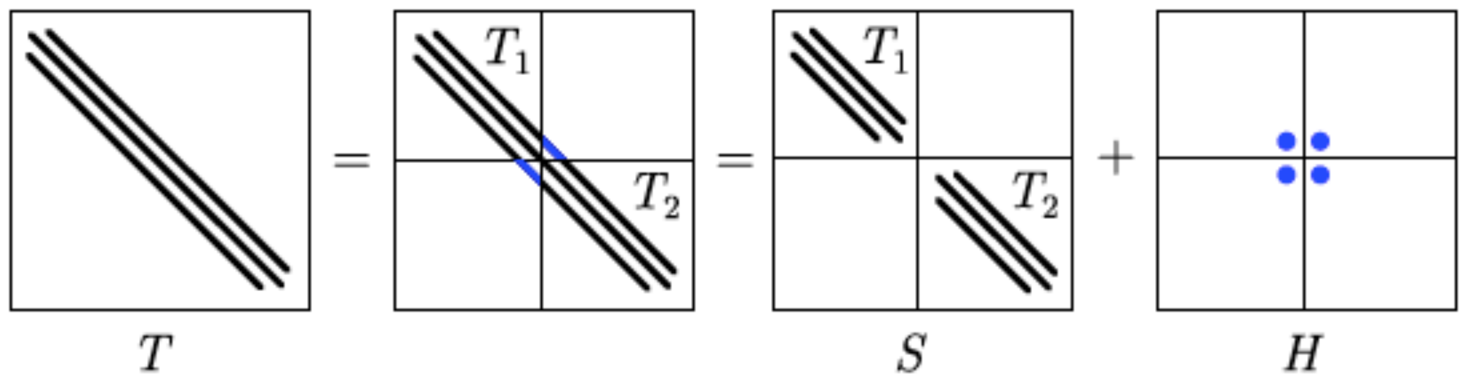
Divide and Conquer algorithm

1. Deflate the eigenvalues and eigenvectors that don't need to be explicitly computed.	*Inherently serial (permutation)*
2. Solve the **secular equation** to compute the eigenvalues.	*Parallelizable*
3. Solve an inverse eigenvalue problem to recover the eigenvectors of the inner system.	*Parallelizable*
4. Recover the eigenvectors of T by computing $Q = RU$, where U has the eigenvectors collected in Stage 3.	*Highly parallelizable (BLAS 3)*
5. Reorder the deflated eigenvalues/eigenvectors into their place.	*Inherently serial (permutation)*

Divide and Conquer algorithm

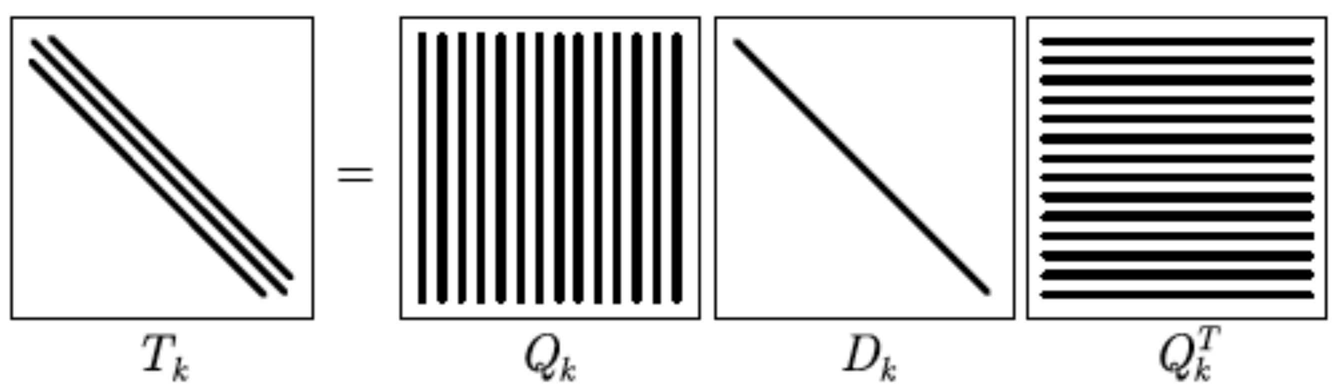
1. Divide

Divide the problem until we reach **base cases**: *k* x *k* tridiagonal systems where *k* is small.



2. Conquer

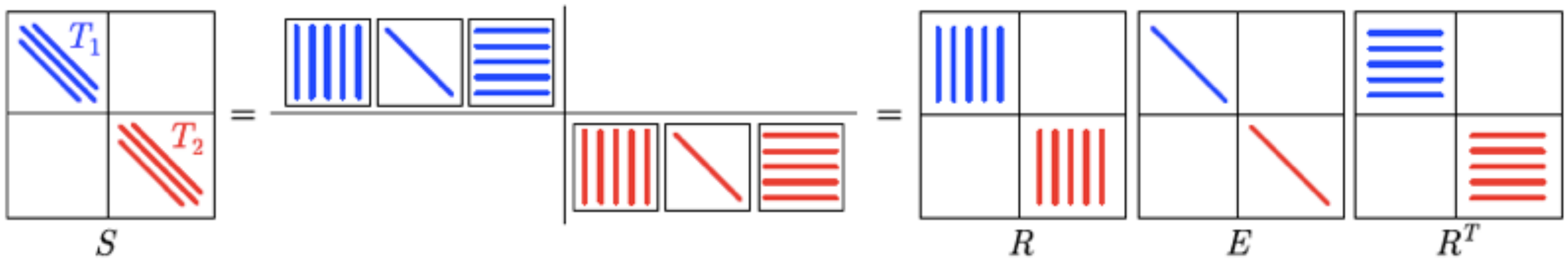
Decompose the base cases using QR.



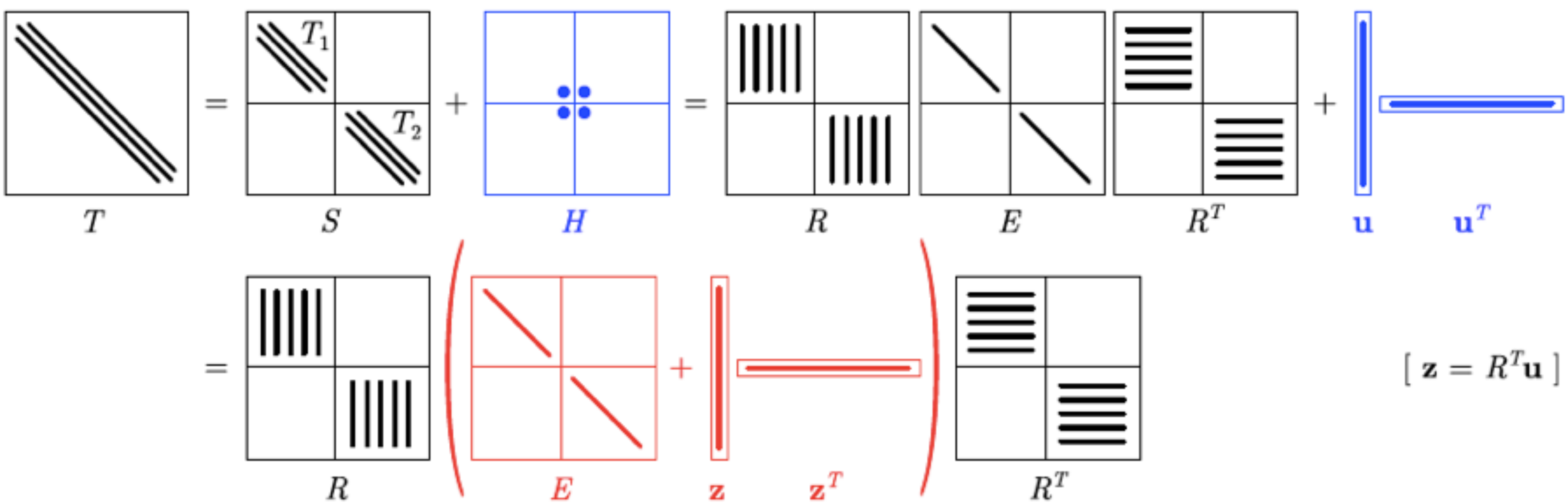
Divide and Conquer algorithm

3. Merge

Build a partial solution *S* from two eigendecompositions.

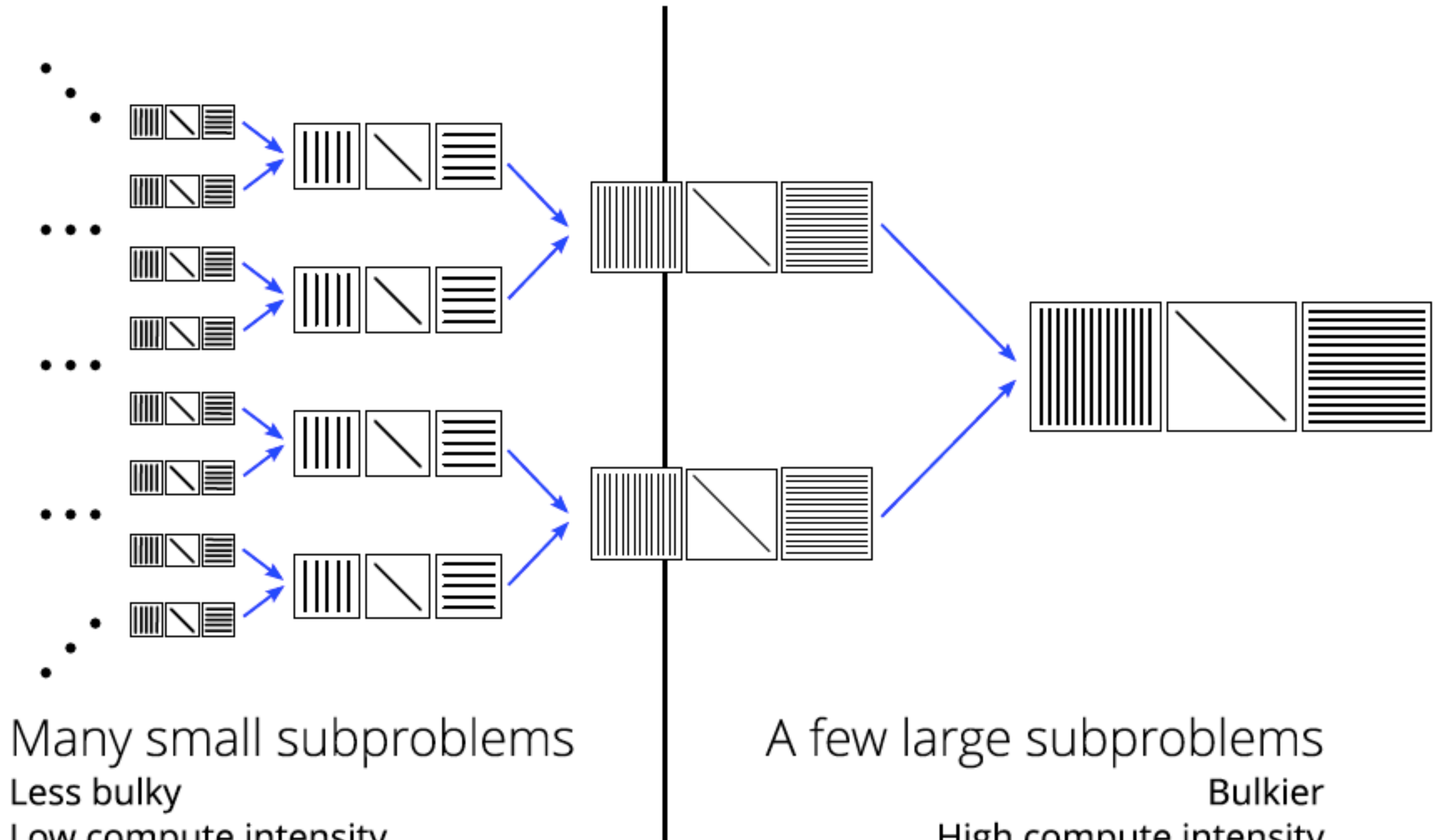


Perform **rank-one update** on *S* to take account of *H*.



Divide and Conquer algorithm

- By the time the algorithm reaches bulky subproblems, it has only a few merge operations to do – the number of subproblems halves at each level.



Timing Results of Latest Code

Some Timings :

On a 1687×1687 $\mathrm{SiOSi_6}$ quantum chemistry matrix,

- Time (Algorithm $\mathbf{MR}^3$) = 5.5 s.
- Time (LAPACK bisection + inverse iteration) = 310 s.
- Time (EISPACK bisection + inverse iteration) = 126 s.
- Time (LAPACK QR) = 1428 s.
- Time (LAPACK Divide & Conquer) = 81 s.

On a 2000×2000 [1,2,1] matrix,

- Time (Algorithm $\mathbf{MR}^3$) = 4.1 s.
- Time (LAPACK bisection + inverse iteration) = 808 s.
- Time (EISPACK bisection + inverse iteration) = 126 s.
- Time (LAPACK QR) = 1642 s.
- Time (LAPACK Divide & Conquer) = 106 s.

numpy.linalg.eigvals

numpy.linalg.eigvals(*a*) [source]

Compute the eigenvalues of a general matrix.

Main difference between eigvals and eig: the eigenvectors aren't returned.

Parameters: **a** : *(..., M, M) array_like*

A complex- or real-valued matrix whose eigenvalues will be computed.

Returns: **w** : *(..., M,) ndarray*

The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

Raises: **LinAlgError**

If the eigenvalue computation does not converge.

See also:

eig	eigenvalues and right eigenvectors of general arrays
eigvalsh	eigenvalues of symmetric or Hermitian arrays.
eigh	eigenvalues and eigenvectors of symmetric/Hermitian arrays.

Notes

New in version 1.8.0.

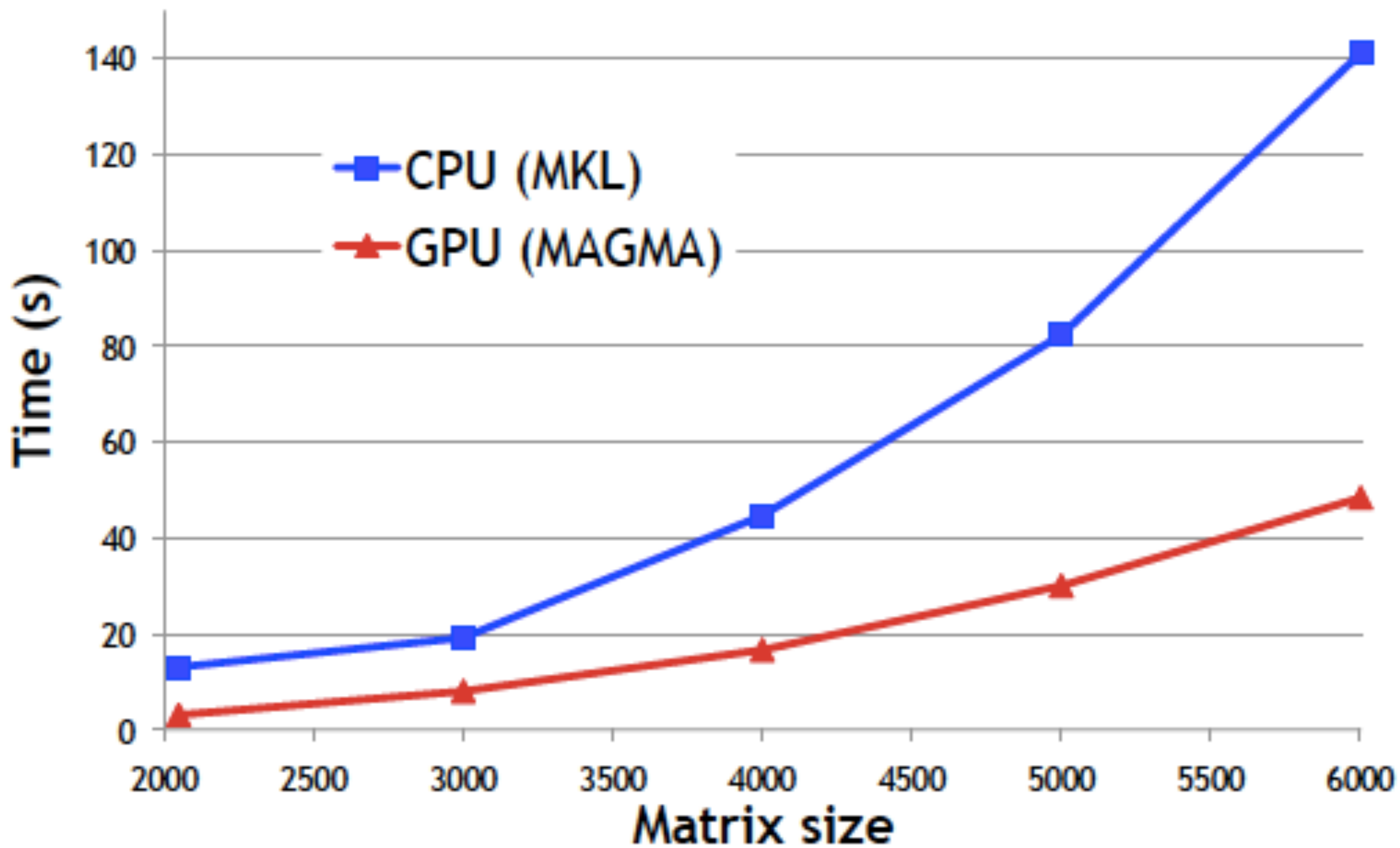
Broadcasting rules apply, see the numpy.linalg documentation for details.

This is implemented using the _geev LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

Complete Eigensolvers

Generalized Hermitian-definite eigenproblem solver ($A x = \lambda B x$)

[double complex arithmetic; based on Divide & Conquer; eigenvalues + eigenvectors]



w/ **Thomas Schulthess** & **Raffaele Solca**
ETH Zurich, Switzerland

GPU	Fermi C2050 [448 CUDA Cores @ 1.15 GHz] + Intel Q9300 [4 cores @ 2.50 GHz]		CPU	AMD ISTANBUL [8 sockets x 6 cores (48 cores) @2.8GHz]
DP peak	515 + 40 GFlop/s		DP peak	538 GFlop/s
System cost ~	$3,000		System cost ~	$30,000
Power * ~	220 W		Power * ~	1,022 W

* Computation consumed power rate (total system rate minus idle rate), measured with *KILL A WATT PS, Model P430*

https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL-ESWorkshop.pdf

Complete Eigensolvers

Hermitian general eigenvalue solver

- Solve $A x = \lambda B x$
- Compute Cholesky factorization of B.
  $B = LL^H$
  - xPOTRF
- Transform the problem to a standard eigenvalue problem
  $A = L^{-1}AL^{-H}$
  - xHEGST
- Solve Hermitian standard Eigenvalue problem
  $A' y = \lambda y$
  - xHEEVx
- Transform back the eigenvectors
  $x = L^{-H} y$
  - xTRSM

https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL-ESWorkshop.pdf

Complete Eigensolvers

Hermitian standard eigenvalue solver

- Solve $A\ y = \lambda\ y$
- Tridiagonalize A

$$T = Q^H A' Q$$

  - xHETRD

- Compute eigenvalues and eigenvectors of the tridiagonal matryx

$$T\ y = \lambda\ y'$$

  - xSTExx

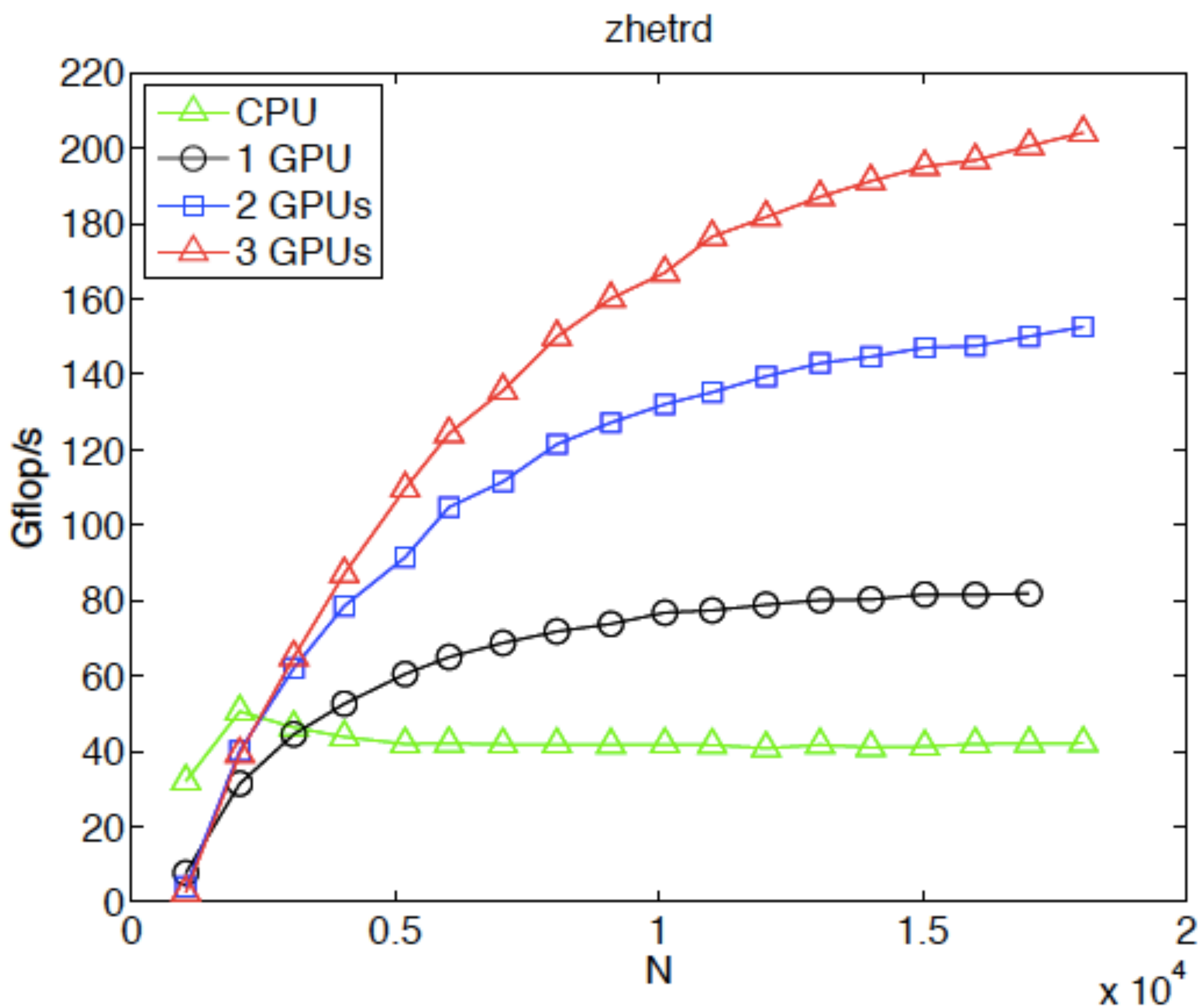
- Transform back the eigenvectors

$$y = Q\ y'$$

  - xUNMTR

https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL-ESWorkshop.pdf

Tridiagonalization on multiGPUs



w/ Ichitaro Yamazaki, UTK
Tingxing Dong, UTK

Keeneland system, using one node
3 NVIDIA GPUs (M2070@ 1.1 GHz, 5.4 GB)
2 **x** 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/ORNL-ESWorkshop.pdf

05/09	Class 9	Dense direct solvers	Understand the principle of LU decomposition and the optimization and parallelization techniques that lead to the LINPACK benchmark.
05/12	Class 10	Dense eigensolvers	Determine eigenvalues and eigenvectors and understand the fast algorithms for diagonalization and orthonormalization.
05/16	Class 11	Sparse direct solvers	Understand reordering in AMD and nested dissection, and fast algorithms such as skyline and multifrontal methods.
05/19	Class 12	Sparse iterative solvers	Understand the notion of positive definiteness, condition number, and the difference between Jacobi, CG, and GMRES.
05/23	Class 13	Preconditioners	Understand how preconditioning affects the condition number and spectral radius, and how that affects the CG method.
05/26	Class 14	Multigrid methods	Understand the role of smoothers, restriction, and prolongation in the V-cycle.
05/30	Class 15	Fast multipole methods, H-matrices	Understand the concept of multipole expansion and low-rank approximation, and the role of the tree structure.