# JDQZ

`jdqz` computes a partial generalized Schur decomposition (or QZ decomposition) of a pair of square matrices or operators.

`Lambda = jdqz(A,B)` and `jdqz(A,B)` return `k` eigenvalues of the matrix pair `(A,B)`, where `k = min(5,n)` and `n = size(A,1)` if `k` has not been specified.

`[X,Jordan] = jdqz(A,B)` returns the eigenvectors `Xx` and the Jordan structure `Jordan`: `A*X = B*X*Jordan`. The diagonal of `Jordan` contains the eigenvalues: `Lambda = diag(Jordan)`. `Jordan` is an `k` by `k` matrix with the eigenvalues on the diagonal and zero or one on the first upper diagonal elements. The other entries are zero.

`[X,Jordan,Q,Z,S,T] = jdqz(A,B)`
If four or more output arguments are required then `Q` and `Z` are `n` by `k` orthonormal, `S` and `T` are `k` by `k` upper triangular such that they form a partial generalized Schur decomposition: `A*Q = Z*S` and `B*Q = Z*T`. Then `Lambda = diag(S)./diag(T)` and `X = Q*Y` with `Y` the eigenvectors of the pair `(S,T)`: `S*Y = T*Y*Jordan` (see also `Options.Schur`).

```
...  = jdqz(A,B)
...  = jdqz('Afun','Bfun')
```
The first input argument is either a square matrix (which can be full or sparse, symmetric or nonsymmetric, real or complex), or a string containing the name of an M-file which applies a linear operator to the columns of a given matrix. In the latter case, the M-file, say `Afun.m`, must return the dimension `n` of the problem with `n = Afun([ ],'dimension')`. For example, `jdqz('fft',...)` is much faster than `jdqz(F,...)`, where `F` is the explicit FFT matrix.
If another input argument is a square `n` by `n` matrix or the name of an M-file, then `B` is this argument (regardless whether `A` is an M-file or a matrix). If `B` has not been specified, then `B` is assumed to be the identity unless `A` is an M-file with two output vectors of dimension `n` with `[Av,Bv] = Afun(v)`, or with `Av = Afun(v,'A')` and `Bv = Afun(v,'B')`.

The remaining input arguments are optional and can be given in practically any order:

```
...  = jdqz(A,B,k,Sigma,Options)
...  = jdqz('Afun','Bfun',k,Sigma,Options),
```

where

| | |
|---|---|
| `k` | an integer, the number of desired eigenvalues. |
| `Sigma` | a scalar shift or a two letter string. |
| `Options` | a structure containing additional parameters. |

If, in addition, there are other input arguments, then they are passed to `'Afun'` and to `'Bfun'` as input arguments(see below).

If `k` is not specified, then `k = min(n,5)` eigenvalues are computed.

If `Sigma` is not specified, then the `k`th eigenvalues largest in magnitude are computed. If `Sigma` is a real or complex scalar, then the `k`th eigenvalues nearest `Sigma` are computed. If `Sigma` is column vector of size `(m,1)`, then the `j`th eigenvalue nearest to `Sigma(min(j,m),1)` is computed for `j = 1:k`. `Sigma` is the "target" for the desired

eigenvalues. If `Sigma` is one of the following strings, then it specifies the desired eigenvalues.

| `Sigma` | Specified eigenvalues |
|---|---|
| `'LM'` | Largest Magnitude |
| `'SM'` | Smallest Magnitude (same as `Sigma = 0`) |
| `'LR'` | Largest Real part |
| `'SR'` | Smallest Real part |
| `'BE'` | Both Ends. Computes `k/2` eigenvalues from each end of the spectrum (one more from the high end if `k` is odd.) |

If `'TestSpace'` is `'Harmonic'` (see `Options`), then `Sigma = 0` is the default, otherwise `Sigma = 'LM'` is the default.

The `Options` structure specifies certain parameters in the algorithm.

| Field name | Parameter | Default |
|---|---|---|
| `Options.Tol` | Convergence tolerance: `norm(r) <= tol/sqrt(k)` | `1e-8` |
| `Options.jmin` | Minimum dimension search subspace `V` | `k+5` |
| `Options.jmax` | Maximum dimension search subspace `V` | `jmin+5` |
| `Options.MaxIt` | Maximum number of iterations. | `100` |
| `Options.v0` | Starting space | `ones+0.1*rand` |
| `Options.Schur` | Gives schur decomposition<br>If `'yes'`, then `X` and `Jordan` are not computed and `[Q,Z,S,T,history]` is the list of output arguments. | `'no'` |
| `Options.TestSpace` | Defines the test subspace `W`<br>`'Standard'`: `W = sigma*A*V+B*V`<br>`'Harmonic'`: `W = A*V-sigma*B*V`<br>`'SearchSpace'`: `W = V`<br>`W = V` is justified if `B` is positive definite. | `'Harmonic'` |
| `Options.Disp` | Shows size of intermediate residuals and the convergence history | `'no'` |
| `Options.LSolver` | Linear solver | `'GMRES'` |
| `Options.LS_Tol` | Residual reduction linear solver | `1,0.7,0.7^2,...` |
| `Options.LS_MaxIt` | Maximum number it. linear solver | `5` |
| `Options.LS_ell` | ell for BiCGstab(ell) | `4` |
| `Options.Precond` | Preconditioner(see below). | identity. |

For instance,

```
Options = struct('Tol',1.0e-8,'LSolver','BiCGstab','LS_ell',4,'Precond',M);
```

changes the convergence tolerance to `1.0e-8`, takes BiCGstab as linear solver, and takes `M` as preconditioner (for ways of defining `M`, see below).

There are a few other `Options` that can be specified. They are listed below .

```
[X,Jordan,history] = jdqr(A,B,...)
[X,Jordan,Q,Z,S,T,history] = jdqz(A,B,...)
```

returns also the convergence history.
`history` is an array of 3 columns: `history(i,1)` is the residual norm at step `j = history(i,2)`, `history(i,3)` is the cumulative number of multiplications by `A` at step `j`. If a search for a new eigenvalue is started at step `J` then `j = history(i,2) = history(i+1,2)`, `history(i,1)` is the norm of the "old" residual, and `history(i+1,1)`

is the norm of the "new" one. `history` is empty if the required number of eigenvalues are detected in the initialization phase.

`jdqz` without input arguments returns the options and its defaults.

### Preconditioning in `jdqz`

The action '`M` inverse' of the preconditioner `M` (an approximation of `A-lambda*B`) on an `n`-vector `v` can be defined in the `Options`

```
Options.Precond
Options.L_Precond          (same as Options.Precond)
Options.U_Precond
Options.P_Precond
```

If no preconditioner has been specified (or is `[ ]`), then `M\v = v` (`M` is the identity).
If `Precond` is an `n` by `n` matrix, say, `K`, then
`M\v = K\v`.
If `Precond` is an `N` by `2*N` matrix, say, `K`, then
`M\v = U\L\v`, where `K = [L,U]`, and `L` and `U` are `n` by `n` matrices.
If `Precond` is a string, say, `'Mi'`, then
if `Mi(v,'L')` and `Mi(v,'U')` return `n`-vectors
`M\v = Mi(Mi(v,'L'),'U')`
otherwise
`M\v = Mi(v)` or `M\v = Mi(v,'preconditioner')`.
Note that `Precond` and `A` can be the same string.
If `L_Precond` and `U_Precond` are strings, say, `'Li'` and `'Ui'`, respectively, then
`M\v = Ui(Li(v))`.
If (`P_precond`,) `L_Precond`, and `U_precond` are `n` by `n` matrices, say, (`P`,) `L`, and `U`, respectively, then
`M\v = U\L\(P*v)` (i.e., `P*M = L*U`).

The way the preconditioner is used can be specified in the `Options`, see below.

### Way of using a preconditioner

The way the preconditioner is used can be specified in the `Options`.

`Options.Type_Precond` (default `'left'`)

The preconditioner can be used as explicit left preconditioner (`'left'`), as explicit right preconditioner (`'right'`), or implicitly (`'impl'`).

In this subsection,
• an MV (matrix vector multiplication) is an operation by `A` together with an operation by `B`,
• a PS (preconditioner solve) is a solution of the system $Mt = v$, where $M$ is the preconditioner (i.e., the computation of `M\v`),

**Explicit versus implicit.** If explicit preconditioning is used, then there is an additional PS needed each time the correction equation is solved. The total number of PSs that `jdqz` will take is equal to the total number of MVs plus the number of Jacobi-Davidson steps (the number of outer iterations).
With implicit preconditioning the number of PSs reduces to the number of MVs plus the number of detected eigenvalues. However, implicit preconditioning requires more memory. For BiCGstab(ell), 2∗ell additional `n`-vectors have to be stored. If GMRES is requested as linear solver, then FGMRES is used, requiring storage of an additional $m$ `n`-vectors. Here $m$ is the maximum number of steps that GMRES needs to achieve the required residual reduction.
By storing the preconditioned vectors of the search subspace the number of PSs can be reduced even further. Then the number of PSs will be equal to the number of MVs. However, this strategy has not been implemented in `jdqz`.
There may be a slight deviation in the count of PSs and MVs if `jdqz` detects more than one eigenpair at the same iteration step.

**Right versus left.** If explicit right preconditioning is used, then the size of the residual is available in the inner loop, that is, the size of the residual of the iterative solver for the correction equation. With explicit left preconditioning, only 'preconditioned' residuals are available. However, right preconditioning, as well as implicit preconditioning, requires slightly more projections.

## Input arguments

`[X,Jordan,Q,Z,S,T,history] = ...`
`jdqz('Afun','Bfun',k,Sigma,Options,Par(1),Par(2),...)`,
The additional input arguments `Par(1), Par(2),...` are passed to `'Afun'` and to `'Bfun'` as input arguments: for instance, `Av = Afun(v,'',Par(1),Par(2),...)`.

With `flag = 'dimension'`, the statement `Afun(v,flag,...)` should return the dimension `n` of the problem. With `flag =''`, `flag = 'A'`, or `flag = 'B'`, it should return an n-vector `Av`; with `flag =''`, it also may return two `n`-vectors: `[Av,Bv]`.

Other choices for `flag` are also allowed (for instance, `flag = 'preconditioner'`, `flag = 'L'`, `flag = 'U'`). However, an error should result for unknown (or unused) selections.

*Example.*

```
function [Av,Bv] = Afun(v,flag,gamma)
  switch flag
    case 'dimension'
      Av = 10000; Bv = []; return
    case ''
    otherwise %%% error message should be included!
      error(sprintf('''%s'' unknown flag',flag))
  end
  [Av,Bv] = ABfun(v,gamma);
return
```

### Additional Options

`Options.Pairs` (default `'no'`)

If `'yes'`, then `jdqz` searches for the complex conjugate eigenpair whenever an eigenpair has been detected. If `A` and `B` are real matrices, or the operators correspond to real matrices, then $(\bar{\lambda}, \bar{x})$ is an eigenpair if $(\lambda, x)$ is one. Since the eigenpairs are not computed in full accuracy and since a generalized Schur decomposition is computed instead of eigenpairs, the conjugate of an approximate eigenpair may not have the required precision and `jdqz` may take additional iterations to obtain the conjugate pair in the desired accuracy.

`Options.Chord` (default `'yes'`)

To balance the influence of $\mathbf{A}$ and $\mathbf{B}$, the eigenvalue problem

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

can be formulated as

$$\theta_2\mathbf{A}\mathbf{x} - \theta_1\mathbf{B}\mathbf{x} = 0$$

where $|\theta_1|^2 + |\theta_2|^2 = 1$ and $\lambda = \theta_1/\theta_2$. In this formulation, the *chordal distance*,

$$\min\{\|(\theta_1, \theta_2) - \zeta(\eta_1, \eta_2)\|_2 \mid \zeta \in \mathbb{C},\ |\zeta| = 1\},$$

is a more natural measure of distance between the eigenvalues $\lambda$ and $\mu$ than the standard measure $|\lambda - \mu|$. Here, $|\eta|^2 + |\eta_2|^2 = 1$ and $\mu \equiv \eta_1/\eta_2$.
With `Options.Chord = 'yes'`, `jdqz` uses the chordal distance to order the approximate eigenvalues in cases `Sigma` is specified as a complex or a real number, or as a pair of complex or real numbers (as target for the desired eigenvalues $\lambda$ or $(\theta_1, \theta_2)$, $\lambda = \theta_1/\theta_2$).

`Options.Scale` (default `1`)

Note that, for $\kappa \neq 0$, the eigenvalue problems $\theta_2\mathbf{A}\mathbf{x} - \theta_1\mathbf{B}\mathbf{x} = 0$ and $(\kappa\theta_2)(\frac{1}{\kappa}\mathbf{A})\mathbf{x} - \theta_1\mathbf{B}\mathbf{x} = 0$ are mathematically equivalent. However, the scaling parameter $\kappa$ affects the chordal distance between (approximate) eigenvectors. Rescaling may lead to a better balance between the two operators $\mathbf{A}$ and $\mathbf{B}$. Instead of scaling the operator $\mathbf{A}$, jdqz applies the scaling to intermediate low dimensional matrices (implicit scaling), which has the same effect, but may save computational costs.

`Options.NSigma` (default `'no'`)

If `'yes'`, then `jdqz` takes as target for the second and following eigenvalues, the best approximate eigenvalues from the current test subspace. Here 'best' is taken with respect to `Sigma`.

`Options.FixShift` (default `'no'`)

If `Options.FixShift` is scalar and `Sigma(1,:)` is a scalar, then `jdqz` takes `Sigma` as shift in the correction equation until the norm of the residual times `Options.FixShift` is less than 1. From then on, the shift is taken equal to the present approximate eigenvalue.
If `Options.FixShift = 'yes'` then `Options.FixShift= 1.0e+3`.
If `Options.FixShift = 'no'` is the same as `Options.FixShift = 0`.

`Options.Track` (default `'1e-4'`)

If the wanted eigenvalue is relatively far from the target, then the algorithm may select approximate eigenvalues that are accidentally close to the target instead of the approximate eigenvalue that is close to the wanted eigenvalue. To avoid this type of misselection, the target can be moved to an approximate eigenvalue that is close to the wanted eigenvalue. The size of the norm of the residual is used to measure the quality of the approximate eigenvalue. If `norm(r)<= Options.track`, then the associated approximate eigenvalue is used as target in the next iteration step.

`Options.AvoidStag` (default `'no'`)

In some situations, the algorithm stagnates because the computed expansion vector for the search subspace belongs to the search subspace or is close to it. With `'yes'`, `jdqz` tries to remedy this type of stagnation. In the correction equation in the next iteration step, `jdqz` projects then on the complete search subspace rather than on the current eigenvector approximation.

`Options.LS_Tol` (default `[1,0.7,0.7`$^2$`,...]`)

`LS_Tol` sets the residual reduction for the linear solver of the correction equation.
If `LS_Tol` is a positive real then the correction equation is solved with a residual reduction of `LS_Tol` in each Jacobi-Davidson step.
If `LS_Tol` is a row $[a_1, a_2, \ldots, a_p]$ of positive reals then the correction equation at the $i$th Jacobi-Davidson step is solved with a residual reduction of $a_i$ provided that $i \leq p$. If $i = p + j$, then the residual reduction at the $i$th Jacobi-Davidson step is $a_p * b^j$ where $b = a_p/a_{p-1}$.
$i$ is reset to 0 when a Schur vector is detected.
The required residual reduction is not obtained if the maximum number `LS_MaxIt` of iteration steps of the linear solver is reached before.
The default for `LS_Tol` is `[1,0.7]` if a preconditioner is used (then $a_i = 0.7^{i-1}$). In the other case, the default is `[0.7,0.49]` (then $a_i = 0.7^i$).

## Accuracy

`Options.Tol` (default `1.0e-8`)

`jdqz` accepts an approximate right Schur vector `q` with associated eigenvalue `(theta1,theta2)` if the 2-norm of the residual `r` is less than `Options.Tol/sqrt(k)`. Then we have that

`norm(A*Q-Z*S,'fro') < Tol` and `norm(B*Q-Z*T,'fro') < Tol`

How accurate the approximations of the eigenvectors and the invariant subspaces are, depends on the conditioning of these quantities.

The residual that `jdqz` computes is given by `r = (I-Z*Z')*(theta2*A*q-theta1*B*q)`.