```julia
using LinearAlgebra, Random, PlutoUI, Quaternions, GenericLinearAlgebra
```

Contents

Fast computations with arrowhead and diagonal-plus-rank-k matrices over associative fields

Quaternions
- Standard form
- Homomorphism

Matrices

Matrix × vector

Inverses
- Arrowhead
- DPRk

Eigenvalue decomposition
- Power method
- RQI and MRQI
- Wielandt's deflation
- Arrowhead matrices
  - Computing the eigenvectors
  - Algorithm
- DPRk matrices
  - Computing the eigenvectors
  - Algorithm

Code
- Quaternions
  - Standard form
- Matrices
- `*()`
- `inv()`
- `Power()`
- `RQI()` and `MRQI()`
- `eigvals()`
- `eigvecs()`
- `eigen()`
- Examples


```julia
TableOfContents(title="Contents", aside=true)
```

Fast computations with arrowhead and diagonal-plus-rank-k matrices over associative fields

by Ivan Slapničar, Thaniporn Chaysri and Nevena Jakovčević Stor

from University of Split, FESB

presented at FoCM, Workshop III.1 *Numerical Linear Algebra*, Paris, June 19-21, 2023.

This work has been fully supported by Croatian Science Foundation under the project IP-2020-02-2240 - http://manaa.fesb.unist.hr.

> We present efficient $O(n^2)$ eigensolvers for arrowhead and DPRk matrices of quaternions. The eigensolvers use a version of Wielandt deflation. Algorithms are elegantly implemented using Julia's polymorphism.

Quaternions

Quaternions are a non-commutative associative number system that extends complex numbers, introduced by Hamilton (1853, 1866). For basic quaternions $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$, the quaternions have the form

$$q = a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k}, \quad a, b, c, d, \in \mathbb{R}.$$

The multiplication table of basic quaternions is the following:

$\times$	$\mathbf{i}$	$\mathbf{j}$	$\mathbf{k}$
$\mathbf{i}$	-1	$\mathbf{k}$	$-\mathbf{j}$
$\mathbf{j}$	$-\mathbf{k}$	-1	$\mathbf{i}$
$\mathbf{k}$	$\mathbf{j}$	$-\mathbf{i}$	-1

Conjugation is given by

$$\bar{q} = a - b\,\mathbf{i} - c\,\mathbf{j} - d\,\mathbf{k}.$$

Then,

$$\bar{q}q = q\bar{q} = |q|^2 = a^2 + b^2 + c^2 + d^2.$$

Let $f(x)$ be a complex analytic function. The value $f(q)$, where $q \in \mathbb{H}$, is computed by evaluating the extension of f to the quaternions at q, see (Sudbery,1979), for example,

$$\sqrt{q} = \pm\left(\sqrt{\frac{\|q\| + a_1}{2}} + \frac{\operatorname{imag}(q)}{\|\operatorname{imag}(q)\|}\sqrt{\frac{\|q\| - a_1}{2}}\right).$$

Basic operations with quaternions and computation of the functions of quaternions are implemented in the package Quaternions.jl.

Standard form

Quaternions p and q are **similar** if

$$\exists x \quad \text{s. t.} \quad p = x^{-1}qx.$$

This is *iff*

$$\operatorname{real}(p) = \operatorname{real}(q) \quad \text{and} \quad \|p\| = \|q\|.$$

The **standard form** of the quaternion q is the (unique) similar quaternion q_s:

$$q_s = x^{-1}qx = a + \hat{b}\,\mathbf{i}, \quad \|x\| = 1, \quad \hat{b} \geq 0,$$

where x is computed as follows:

if $c=d=0$, then $x=1$,

if $b<0$, then $x=-\mathbf{j}$, ortherwise,

if $c^2+d^2>0$, then $x=\hat x/\|\hat x\|$, where $\hat x = \|\operatorname{\mathbf{imag}}(q)\| + b - d\,\mathbf{j} + c\,\mathbf{k}$.

Homomorphism

Quaternions are homomorphic to $\mathbb{C}^{2\times 2}$:

$$q \to \begin{bmatrix} a+b\,\mathbf{i} & c+d\,\mathbf{i} \\ -c+d\,\mathbf{i} & a-b\,\mathbf{i} \end{bmatrix} \equiv C(q),$$

with eigenvalues q_s and $\bar q_s$.

Matrices

All matrices are in $\mathbb{F}^{n\times n}$ where $\mathbb{F}\in\{\mathbb{R},\mathbb{C},\mathbb{H}\}$. $\mathbb{H}$ is a non-commutative field of quaternions.

Arrowhead matrix (**Arrow**) is a matrix of the form

$$A = \begin{bmatrix} D & u \\ v^* & \alpha \end{bmatrix},$$

where

$$\operatorname{diag}(D), u, v \in \mathbb{F}^{n-1}, \quad \alpha\in\mathbb{F},$$

or any symmetric permutation of such a matrix.

Diagonal-plus-rank-k matrix (**DPRk**) is a matrix of the form

$$A = \Delta + x\rho y^*$$

where

$$\operatorname{diag}(\Delta)\in\mathbb{F}^n, \quad x,y\in\mathbb{F}^{n\times k}, \quad \rho\in\mathbb{F}^{k\times k}.$$

Matrix × vector

Products

$$w = Az$$

are computed in $O(n)$ operations.

Let $A = \operatorname{Arrow}(D, u, v, \alpha)$. Then

$$\begin{aligned}
w_j &= d_j z_j + u_j z_i, \quad j = 1, 2, \cdots, i-1 \\
w_i &= v_{1:i-1}^* z_{1:i-1} + \alpha z_i + v_{i:n-1}^* z_{i+1:n} \\
w_j &= u_{j-1} z_i + d_{j-1} z_j, \quad j = i+1, i+2, \cdots, n.
\end{aligned}$$

Let $A = \operatorname{DPRk}(\Delta, x, y, \rho)$ and let $\beta = \rho(y^* z)$. Then

$$w_i = \delta_i z_i + x_i \beta, \quad i = 1, 2, \cdots, n.$$

Inverses

Inverses are computed in $O(n)$ operations.

Arrowhead

Let $A = \mathrm{Arrow}(D,u,v,\alpha)$ be nonsingular.

Let P be the permutation matrix of the permutation $p = (1,2,\cdots,i-1,n,i,i+1,\cdots,n-1)$.

If all $d_j \neq 0$, the inverse of A is a DPRk (DPR1) matrix

$$A^{-1} = \Delta + x\rho y^*,$$

where

$$\Delta = P\begin{bmatrix} D^{-1} & 0\\ 0 & 0\end{bmatrix}P^T, \quad x = P\begin{bmatrix} D^{-1}u\\ -1\end{bmatrix}\rho, \quad y = P\begin{bmatrix} D^{-\star}v\\ -1\end{bmatrix}, \quad \rho = (\alpha - v^\star D^{-1}u)^{-1}.$$

If $d_j = 0$, the inverse of A is an Arrow with the tip of the arrow at position (j,j) and zero at position A_{ii} (the tip and the zero on the shaft change places). In particular, let $\hat P$ be the permutation matrix of the permutation $\hat p = (1,2,\cdots,j-1,n,j,j+1,\cdots,n-1)$. Partition D, u and v as

$$D = \begin{bmatrix} D_1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & D_2\end{bmatrix}, \quad u = \begin{bmatrix} u_1\\ u_j\\ u_2\end{bmatrix}, \quad v = \begin{bmatrix} v_1\\ v_j\\ v_2\end{bmatrix}.$$

Then

$$A^{-1} = P\begin{bmatrix} \hat D & \hat u\\ \hat v^* & \hat\alpha\end{bmatrix}P^T,$$

where

$$\hat D = \begin{bmatrix} D_1^{-1} & 0 & 0\\ 0 & D_2^{-1} & 0\\ 0 & 0 & 0\end{bmatrix}, \quad \hat u = \begin{bmatrix} -D_1^{-1}u_1\\ -D_2^{-1}u_2\\ 1\end{bmatrix}u_j^{-1}, \quad \hat v = \begin{bmatrix} -D_1^{-\star}v_1\\ -D_2^{-\star}v_2\\ 1\end{bmatrix}v_j^{-1},$$
$$\hat\alpha = v_j^{-\star}\left(-\alpha + v_1^\star D_1^{-1}u_1 + v_2^\star D_2^{-1}u_2\right)u_j^{-1}.$$

DPRk

Let $A = \mathrm{DPRk}(\Delta,x,y,\rho)$ be nonsingular.

If all $\delta_j \neq 0$, the inverse of A is a DPRk matrix

$$A^{-1} = \hat\Delta + \hat x\hat\rho\hat y^*,$$

where

$$\hat{\Delta}=\Delta^{-1}, \qquad \hat{x}=\Delta^{-1}x, \quad \hat{y}=\Delta^{-*}y, \quad \hat{\rho}=-\rho(I-y^*\Delta^{-1}x\rho)^{-1}.$$

If $k=1$ and $\delta_j=0$, the inverse of A is an arrowhead matrix with the tip of the arrow at position (j,j). In particular, let P be the permutation matrix of the permutation $p=(1,2,\cdots,j-1,n,j,j+1,\cdots,n-1)$. Partition Δ, x and y as

$$\Delta=\begin{bmatrix}\Delta_1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & \Delta_2\end{bmatrix}, \quad x=\begin{bmatrix}x_1\\ x_j\\ x_2\end{bmatrix}, \quad y=\begin{bmatrix}y_1\\ y_j\\ y_2\end{bmatrix}.$$

Then,

$$A^{-1}=P\begin{bmatrix}D & u\\ v^* & \alpha\end{bmatrix}P^T,$$

where

$$D=\begin{bmatrix}\Delta_1^{-1} & 0\\ 0 & \Delta_2^{-1}\end{bmatrix}, \quad u=\begin{bmatrix}-\Delta_1^{-1}x_1\\ -\Delta_2^{-1}x_2\end{bmatrix}x_j^{-1}, \quad v=\begin{bmatrix}-\Delta_1^{-\star}y_1\\ -\Delta_2^{-\star}y_2\end{bmatrix}y_j^{-1},$$
$$\alpha=(y_j^{-1})^\star\left(\rho^{-1}+y_1^\star\Delta_1^{-1}x_1+y_2^\star\Delta_2^{-1}x_2\right)x_j^{-1}.$$

Eigenvalue decomposition

Right eigenpairs (λ, x) of a quaternionic matrix satisfy

$$Ax=x\lambda, \quad x\neq 0.$$

Usually, x is chosen such that λ is the standard form.

Eigenvalues are invariant under similarity.

> Eigenvalues are **NOT** shift invariant, that is, eigenvalues of the shifted matrix are **NOT** the shifted eigenvalues.

If λ is in the standard form, it is invariant under similarity with complex numbers.

Power method

The power method produces a sequence of vectors

$$y_k = Ax_k, \quad x_{k+1} = \frac{y_k}{\|y_k\|}, \quad k = 0, 1, 2, \ldots$$

If λ is a dominant eigenvalue, and x_0 has a component in the direction of its unit eigenvector x, then $x_k \to x$ and $x^* Ax = \lambda$. The convergence is linear.

> If A is an arrowhead or a DPRk matrix, then, due to fast matrix × vector multiplication, one step of the method requires $O(n)$ operations.

RQI and MRQI

The **Rayleigh Quotient Iteration** (RQI) produces sequences of shifts and vectors

$$\mu_k = \frac{1}{x_k^* x_k} x_k^* A x_k, \quad y_k = (A - \mu_k I)^{-1} x_k, \quad x_{k+1} = \frac{y_k}{\|y_k\|}, \quad k = 0, 1, 2, \ldots$$

The **Modified Rayleigh Quotient Iteration** (MRQI) produces sequences of shifts and vectors

$$\mu_k = \frac{1}{x^T x} x^T A x, \quad y_k = (A - \mu_k I)^{-1} x_k, \quad x_{k+1} = \frac{y_k}{\|y_k\|}, \quad k = 0, 1, 2, \ldots$$

Since the eigenvalues are not shift invariant, only real shifts can be used. However, this works fine due to the following: let the matrix $A - \mu I$ have purely imaginary standard eigenvalue:

$$(A - \mu I)x = x(i\lambda), \qquad \mu, \lambda \in \mathbb{R}.$$

Then

$$Ax = \mu x + xi\lambda = x(\mu + i\lambda).$$

> If A is an arrowhead or a DPRk matrix, then, due to fast inverses, one step of the methods requires $O(n)$ operations.

Wielandt's deflation

- Let A be a (real, complex, or quaternionic) matrix.
- Let (λ, u) be a right eigenpair of A.
- Choose z such that $z^*u = 1$, say $z^* = [1/u_1 \quad 0 \quad \cdots \quad 0]$.
- Compute the deflated matrix $\tilde{A} = (I - uz^*)A$.
- Then $(0, u)$ is an eigenpair of $\tilde{A}$.
- Further, if (μ, v) is an eigenpair of A, then $(\mu, \tilde{v})$, where $\tilde{v} = (I - uz^*)v$ is an eigenpair of $\tilde{A}$.

Proofs: Using $Au = u\lambda$ and $z^*u = 1$, the first statement holds since

$$\tilde{A}u = (I - uz^*)Au = Au - uz^*Au = u\lambda - uz^*u\lambda = 0.$$

Further,

$$\begin{aligned}
\tilde{A}\tilde{v} &= (I - uz^*)A(I - uz^*)v \\
&= (I - uz^*)Av - Auz^*v + uz^*Auz^*v \\
&= (I - uz^*)v\mu - u\lambda z^*v + uz^*u\lambda z^*v \\
&= \tilde{v}\mu
\end{aligned}.$$

Arrowhead matrices

Lemma 1. Let A be an arrowhead matrix partitioned as

$$A = \begin{bmatrix} \delta & 0 & \chi \\ 0 & \Delta & x \\ \bar{v} & y^* & \alpha \end{bmatrix},$$

where χ, v and α are scalars, x and y are vectors, and Δ is a diagonal matrix.

Let $\left(\lambda, \begin{bmatrix} \nu \\ u \\ \psi \end{bmatrix}\right)$, where ν and ψ are scalars, and u is a vector, be an eigenpair of A. Then, the deflated matrix $\tilde{A}$ has the form

$$\tilde{A} = \begin{bmatrix} 0 & 0^T \\ w & \hat{A} \end{bmatrix}, \tag{1}$$

where

$$w = \begin{bmatrix} -u\frac{1}{\nu}\delta \\ -\psi\frac{1}{\nu}\delta + \bar{v} \end{bmatrix},$$

and $\hat{A}$ is an arrowhead matrix

$$\hat{A} = \begin{bmatrix} \Delta & -u\frac{1}{\nu}\chi + x \\ y^* & -\psi\frac{1}{\nu}\chi + \alpha \end{bmatrix}. \tag{2}$$

Proof: We have

$$\begin{aligned} \tilde{A} &= \left(\begin{bmatrix} 1 & 0^T & 0 \\ 0 & I & 0 \\ 0 & 0^T & 1 \end{bmatrix} - \begin{bmatrix} \nu \\ u \\ \psi \end{bmatrix} \begin{bmatrix} \frac{1}{\nu} & 0^T & 0 \end{bmatrix} \right) \begin{bmatrix} \delta & 0 & \chi \\ 0 & \Delta & x \\ \bar{v} & y^* & \alpha \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu} & I & 0 \\ -\psi\frac{1}{\nu} & 0^T & 1 \end{bmatrix} \begin{bmatrix} \delta & 0 & \chi \\ 0 & \Delta & x \\ \bar{v} & y^* & \alpha \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu}\delta & \Delta & -u\frac{1}{\nu}\chi + x \\ -\psi\frac{1}{\nu}\delta + \bar{v} & y^* & -\psi\frac{1}{\nu}\chi + \alpha \end{bmatrix}, \end{aligned} \tag{3}$$

as desired. □

Lemma 2. Let A and $\hat{A}$ be as in Lemma 4. If $\left(\mu, \begin{bmatrix} \hat{z} \\ \hat{\xi} \end{bmatrix}\right)$ is an eigenpair of $\hat{A}$, then the eigenpair of A is

$$\left(\mu, \begin{bmatrix} \zeta \\ \hat{z} + u\frac{1}{\nu}\zeta \\ \hat{\xi} + \psi\frac{1}{\nu}\zeta \end{bmatrix}\right), \tag{4}$$

where ζ is the solution of the scalar Sylvester equation

$$\left(\delta + \chi\psi\frac{1}{\nu}\right)\zeta - \zeta\mu = -\chi\hat{\xi}. \tag{5}$$

Proof: If μ is an eigenvalue of $\hat{A}$, it is obviously also an eigenvalue of $\tilde{A}$, and then also of A. Assume that the corresponding eigenvector of A is partitioned as $\begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix}$. By combining (1), (2) and (3), we have

$$\begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu}\delta & \Delta & -u\frac{1}{\nu}\chi + x \\ -\psi\frac{1}{\nu}\delta + \bar{v} & y^* & -\psi\frac{1}{\nu}\chi + \alpha \end{bmatrix} \begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu} & I & 0 \\ -\psi\frac{1}{\nu} & 0^T & 1 \end{bmatrix} \begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix} = \begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu} & I & 0 \\ -\psi\frac{1}{\nu} & 0^T & 1 \end{bmatrix} \begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix} \mu,$$

or

$$\begin{bmatrix} 0 & 0^T & 0 \\ -u\frac{1}{\nu}\delta & \Delta & -u\frac{1}{\nu}\chi + x \\ -\psi\frac{1}{\nu}\delta + \bar{v} & y^* & -\psi\frac{1}{\nu}\chi + \alpha \end{bmatrix} \begin{bmatrix} 0 \\ -u\frac{1}{\nu}\zeta + z \\ -\psi\frac{1}{\nu}\zeta + \xi \end{bmatrix} = \begin{bmatrix} 0 \\ -u\frac{1}{\nu}\zeta + z \\ -\psi\frac{1}{\nu}\zeta + \xi \end{bmatrix} \mu,$$

Since the bottom right 2×2 matrix is $\hat{A}$, the above equation implies

$$\begin{bmatrix} \hat{z} \\ \hat{\xi} \end{bmatrix} = \begin{bmatrix} -u\frac{1}{\nu}\zeta + z \\ -\psi\frac{1}{\nu}\zeta + \xi \end{bmatrix},$$

which proves (4). It remains to compute ζ. The first component of the equality

$$A \begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix} = \begin{bmatrix} \delta & 0 & \chi \\ 0 & \Delta & x \\ \bar{v} & y^* & \alpha \end{bmatrix} \begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix} = \begin{bmatrix} \zeta \\ z \\ \xi \end{bmatrix} \mu$$

implies

$$\delta\zeta + \chi\xi = \zeta\mu,$$

or

$$\delta\zeta + \chi(\hat{\xi} + \psi\frac{1}{\nu}\zeta) = \zeta\mu,$$

which is exactly the equation (5). □

Computing the eigenvectors

Let $\left(\lambda, \begin{bmatrix} \nu \\ u \\ \psi \end{bmatrix}\right)$ be an eigenpair of the matrix A, that is

$$\begin{bmatrix} \delta & 0 & \chi \\ 0 & \Delta & x \\ \bar{\upsilon} & y^* & \alpha \end{bmatrix} \begin{bmatrix} \nu \\ u \\ \psi \end{bmatrix} = \begin{bmatrix} \nu \\ u \\ \psi \end{bmatrix} \lambda.$$

If λ and ψ are known, then the other components of the eigenvector are solutions of scalar Sylvester equations

$$\delta\nu - \nu\lambda = -\chi\psi, \tag{6}$$

$$\Delta_{ii} u_i - u_i \lambda = -x_i \psi, \quad i = 1, \ldots, n-2.$$

By setting

$$\gamma = \delta + \chi\psi \frac{1}{\nu}$$

the Sylvester equation (5) becomes

$$\gamma\zeta - \zeta\mu = -\chi\hat{\xi}. \tag{7}$$

Dividing (6) by ν from the right gives

$$\gamma = \nu\lambda \frac{1}{\nu}. \tag{8}$$

Algorithm

In the first (forward) pass, in each step the absolutely largest eigenvalue and its eigenvector are computed by the power method. The first element of the current vector x and the the first and the last elements of the current eigenvector are stored. The current value γ is computed using (8) and stored. The deflation is then performed according to Lemma 1.

The eigenvectors are reconstructed bottom-up, that is from the smallest matrix to the original one (a backward pass). In each iteration we need to have the access to the first element of the vector x which was used to define the current Arrow matrix, its absolutely largest eigenvalue, and the first and the last elements of the corresponding eigenvector.

In the ith step, for each $j = i+1, \ldots, n$ the following steps are performed:

1. The equation (5) is solved for ζ (the first element of the eigenvector of the larger matrix). The quantity $\hat{\xi}$ is the last element of the eigenvectors and was stored in the forward pass.
2. The first element of eigenvector of super-matrix is updated (set to ζ).
3. The last element of the eigenvectors of the super matrix is updated using (4).

Iterations are completed in $O(n^2)$ operations.

After all iterations are completed, we have:

- the absolutely largest eigenvalue and its eigenvector (unchanged from the first run of the chosen eigensolver),
- all other eigenvalues and the last elements of their corresponding eigenvectors.

The rest of the elements of the remaining eigenvectors are computed using the procedure described at the beginning of the previous section. This step also requires $O(n^2)$ operations.

DPRk matrices

Lemma 3. Let A be a DPRk matrix partitioned as

$$A = \begin{bmatrix} \delta & 0^T \\ 0 & \Delta \end{bmatrix} + \begin{bmatrix} \chi \\ x \end{bmatrix} \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix}.$$

Let $\left(\lambda, \begin{bmatrix} \nu \\ u \end{bmatrix}\right)$ be the eigenpair of A. Then, the deflated matrix $\tilde{A}$ has the form

$$\tilde{A} = \begin{bmatrix} 0 & 0^T \\ w & \hat{A} \end{bmatrix}, \tag{9}$$

where

$$w = -u\frac{1}{\nu}\delta - u\frac{1}{\nu}\chi\rho\bar{v} + x\rho\bar{v}$$

and $\hat{A}$ is a DPRk matrix

$$\hat{A} = \Delta + \hat{x}\rho y^*, \quad \hat{x} = x - u\frac{1}{\nu}\chi. \tag{10}$$

Proof: We have

$$\begin{aligned} \tilde{A} &= \left(\begin{bmatrix} 1 & 0^T \\ 0 & I \end{bmatrix} - \begin{bmatrix} \nu \\ u \end{bmatrix} \begin{bmatrix} \frac{1}{\nu} & 0^T \end{bmatrix} \right) \left(\begin{bmatrix} \delta & 0^T \\ 0 & \Delta \end{bmatrix} + \begin{bmatrix} \chi \\ x \end{bmatrix} \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix} \right) \\ &= \begin{bmatrix} 0 & 0^T \\ -u\frac{1}{\nu} & I \end{bmatrix} \cdot \begin{bmatrix} \delta + \chi\rho\bar{v} & \chi\rho y^* \\ x\rho\bar{v} & \Delta + x\rho y^* \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0^T \\ -u\frac{1}{\nu}\delta - u\frac{1}{\nu}\chi\rho\bar{v} + x\rho\bar{v} & -u\frac{1}{\nu}\chi\rho y^* + \Delta + x\rho y^* \end{bmatrix}, \end{aligned} \tag{11}$$

as desired. □

Lemma 4. Let A, $\tilde{A}$, and $\hat{A}$ be as in Lemma 1. If $(\mu, \hat{z})$ is an eigenpair of $\hat{A}$, then the eigenpair of A is

$$\left(\mu, \begin{bmatrix} \zeta \\ \hat{z} + u\frac{1}{\nu}\zeta \end{bmatrix}\right),$$

where ζ is the solution of the Sylvester equation

$$(\delta + \chi\rho\bar{v} + \chi\rho y^* u\frac{1}{\nu})\zeta - \zeta\mu = -\chi\rho y^* \hat{z}. \tag{12}$$

Proof: If μ is an eigenvalue of $\hat{A}$, it is obviously also an eigenvalue of $\tilde{A}$, and then also of A. Assume that the corresponding eigenvector of A is partitioned as $\begin{bmatrix} \zeta \\ z \end{bmatrix}$. By combining (9) and (11) and the previous results, it must hold

$$\begin{bmatrix} 0 & 0^T \\ w & \hat{A} \end{bmatrix} \begin{bmatrix} 0 & 0^T \\ -u\frac{1}{\nu} & I \end{bmatrix} \begin{bmatrix} \zeta \\ z \end{bmatrix} = \begin{bmatrix} 0 & 0^T \\ -u\frac{1}{\nu} & I \end{bmatrix} \begin{bmatrix} \zeta \\ z \end{bmatrix} \mu$$

or

$$\begin{bmatrix} 0 & 0^T \\ w & \hat{A} \end{bmatrix} \begin{bmatrix} 0 \\ -u\frac{1}{\nu}\zeta + z \end{bmatrix} = \begin{bmatrix} 0 \\ -u\frac{1}{\nu}\zeta + z \end{bmatrix} \mu.$$

Therefore, $\hat{z} = -u\frac{1}{\nu}\zeta + z$, or

$$z = \hat{z} + u\frac{1}{\nu}\zeta, \tag{13}$$

and it remains to compute ζ. From the equality

$$\left(\begin{bmatrix} \delta & 0^T \\ 0 & \Delta \end{bmatrix} + \begin{bmatrix} \chi \\ x \end{bmatrix} \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix}\right) \begin{bmatrix} \zeta \\ z \end{bmatrix} = \begin{bmatrix} \zeta \\ z \end{bmatrix} \mu$$

it follows

$$\begin{bmatrix} \delta + \chi\rho\bar{v} & \chi\rho y^* \\ x\rho\bar{v} & \Delta + x\rho y^* \end{bmatrix} \begin{bmatrix} \zeta \\ z \end{bmatrix} = \begin{bmatrix} \zeta \\ z \end{bmatrix} \mu.$$

Equating the first elements and using (13) gives

$$(\delta + \chi\rho\bar{v})\zeta + \chi\rho y^* \hat{z} + \chi\rho y^* u\frac{1}{\nu}\zeta = \zeta\mu,$$

which is exactly the Sylvester equation (12). $\square$

Computing the eigenvectors

Let the DPRk matrix A and its eigenpair be defined as in Lemma 3. Let x be partitioned row-wise as

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}.$$

Set $\alpha = \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix} \begin{bmatrix} \nu \\ u \end{bmatrix}$. From

$$\left(\begin{bmatrix} \delta & 0^T \\ 0 & \Delta \end{bmatrix} + \begin{bmatrix} \chi \\ x \end{bmatrix} \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix} \right) \begin{bmatrix} \nu \\ u \end{bmatrix} = \begin{bmatrix} \nu \\ u \end{bmatrix} \lambda.$$

it follows that the elements of the eigenvector satisfy scalar Sylvester equations

$$\begin{aligned} \delta\nu - \nu\lambda &= -\chi\alpha, \\ \Delta_{ii} u_i - u_i \lambda &= -x_i \alpha, \quad i = 1, \ldots, n-1. \end{aligned} \tag{14}$$

If λ, ν and the first $k-1$ components of u are known, then α is computed from the first k equations in (14), that is, by solving the system

$$\begin{bmatrix} \chi \\ x_1 \\ \vdots \\ x_{k-1} \end{bmatrix} \alpha = \begin{bmatrix} \nu\lambda - \delta\nu \\ u_1 \lambda - \Delta_{11} u_1 \\ \vdots \\ u_{k-1}\lambda - \Delta_{k-1,k-1} u_{k-1} \end{bmatrix},$$

and u_i, $i = k, \ldots, n-1$, are computed by solving the remaining Sylvester equations in (14).

Lemma 5. Assume A and its eigenpair are given as in Lemma 3, and $\hat{A}$ and its eigenpair are given as in Lemma 4. Set in (12)

$$\gamma = \delta + \chi\rho\bar{v} + \chi\rho y^* u\frac{1}{\nu}, \quad \alpha = \rho y^* \hat{z}.$$

Then,

$$\gamma = \nu\lambda\frac{1}{\nu}, \tag{15}$$

and α is the solution of the system

$$\begin{bmatrix} x_1 - u_1\frac{1}{\nu}\chi \\ \vdots \\ x_k - u_k\frac{1}{\nu}\chi \end{bmatrix} \alpha = \begin{bmatrix} \hat{z}_1\mu - \Delta_{11}\hat{z}_1 \\ \vdots \\ \hat{z}_k\mu - \Delta_{kk}\hat{z}_k \end{bmatrix}. \tag{16}$$

Proof: The formula for γ follows by multiplying the first elements of the equation

$$\left(\begin{bmatrix} \delta & 0^T \\ 0 & \Delta \end{bmatrix} + \begin{bmatrix} \chi \\ x \end{bmatrix} \rho \begin{bmatrix} \bar{v} & y^* \end{bmatrix} \right) \begin{bmatrix} \nu \\ u \end{bmatrix} = \begin{bmatrix} \nu \\ u \end{bmatrix} \lambda$$

with $\frac{1}{\nu}$ from the right.

Consider the equation $\hat{A}\hat{z} = \hat{z}\mu$, that is,

$$[\Delta + (x - u\frac{1}{\nu}\chi)\rho y^*]\hat{z} = \hat{z}\mu.$$

The i-th component is

$$\Delta_{ii}\hat{z}_i + (x_i - u_i\frac{1}{\nu}\chi)\alpha = \hat{z}_i\mu,$$

which gives (16). ☐

Algorithm

Lemmas 3 and 5 are used as follows. In the first (forward) pass, in each step the absolutely largest eigenvalue and its eigenvector are computed by the power method. The first two elements of the current vector x and the current eigenvector are stored. The current value γ is computed using (15) and stored. The deflation is then performed according to Lemma 3.

The eigenvectors are reconstructed bottom-up, that is from the smallest 2×2 matrix to the original one (a backward pass). In each iteration we need to have the access to the first two elements of the vector x which was used to define the current DPRk matrix, its absolutely largest eigenvalue, and the first two elements of the corresponding eigenvector.

In the ith step, for each $j = i+1, \ldots, n$, the following steps are performed:

1. The value α is computed from (16).
2. The equation (12), which now reads $\gamma\zeta - \zeta\mu = -\chi\alpha$ is solved for ζ (the first element of the eigenvector of the larger matrix).
3. First element of eigenvector of super-matrix is updated (set to ζ).

Iterations are completed in $O(n^2)$ operations.

After all iterations are completed, we have:

- the first computed eigenvalue and its eigenvector (unchanged from the first run of the eigensolver of choice),
- all other eigenvalues and the first elements of their corresponding eigenvectors.

The rest of the elements of the remaining eigenvectors are computed using the procedure described at the beginning of the section. This step also requires $O(n^2)$ operations.

Code

Quaternions

```
unblock (generic function with 1 method)
```

```julia
begin
    import Base: eps, imag
    eps(::Type{Complex{T}}) where T=eps(T)
    eps(::Type{Quaternions.Quaternion{T}}) where T=eps(T)
    const QuaternionF64=Quaternion{Float64}
    im=Quaternion(0,1,0,0)
    jm=Quaternion(0,0,1,0)
    km=Quaternion(0,0,0,1)
    Quaternion{T}(x::Complex) where {T<:Real} =
    Quaternion(convert(T,real(x)),convert(T,imag(x)),0,0)
    imag(q::Quaternion)=(q-conj(q))/2

    # Quaternion to 2x2 complex
    function q2c(c::T) where T<:QuaternionF64
        return [complex(c.s, c.v1) complex(c.v2,c.v3);
                complex(-c.v2,c.v3) complex(c.s,-c.v1)]
    end
    # For compatibility
    function q2c(c::T) where T
        return c
    end

    # Converts block matrix to ordinary matrix
    unblock(A) = mapreduce(identity, hcat, [mapreduce(identity, vcat, A[:,i])
        for i = 1:size(A,2)])
end
```

```
1.1102230246251565e-16
```

```julia
begin
    # Test the arithmetic
    a=randn(QuaternionF64)
    b=√a
    abs(b*b-a)
end
```

Standard form

```
standardformx (generic function with 4 methods)
```

```
begin
    function standardformx(a::Vector{T}) where T<:QuaternionF64
        # Computes vector x such that inv.(x).*a .*x is in the standard form
        n=length(a)
        x=Array{T}(undef,n)
        for i=1:n
            x[i]=standardformx(a[i])
        end
        x
    end

    function standardformx(a::QuaternionF64)
        # Return standard form of a
        b=copy(a)
        if norm([b.v2 b.v3])>0.0
            x=norm(Quaternions.imag(b))+b.v1-b.v3*jm+b.v2*km
            x/=abs(x)
        elseif b.v1<0.0
            x=-jm
        else
            x=1.0
        end
        return x
    end

    function standardform(a::QuaternionF64)
        # Return standard form of a
        x=standardformx(a)
        # watch out for the correct division: / and not \
        return (x\a)*x
    end

    standardform(a::Float64)=a
    standardformx(a::Float64)=one(a)
    standardform(a::ComplexF64)=a
    standardformx(a::ComplexF64)=one(a)
end
```

Matrices

```
begin
    # Structures
    struct Arrow{T} <: AbstractMatrix{T}
        D::AbstractVector{T}
        u::AbstractVecOrMat{T}
        v::AbstractVecOrMat{T}
        α
        i::Int
    end

    struct DPRk{T} <: AbstractMatrix{T}
        Δ::AbstractVector{T}
        x::AbstractVecOrMat{T}
        y::AbstractVecOrMat{T}
        ρ
    end
end
```

```
adjoint (generic function with 52 methods)
```

```julia
begin
    import Base: size, getindex
    import  LinearAlgebra: Matrix, adjoint, transpose

    # Arrowhead
    size(A::Arrow, dim::Integer) = length(A.D)+1
    size(A::Arrow)= size(A,1), size(A,1)

    function getindex(A::Arrow,i::Integer,j::Integer)
        n=size(A,1)
        if i==j<A.i; return A.D[i]
        elseif i==j>A.i; return A.D[i-1]
        elseif i==j==A.i; return A.α
        elseif i==A.i&&j<A.i; return adjoint(A.v[j])
        elseif i==A.i&&j>A.i; return adjoint(A.v[j-1])
        elseif j==A.i&&i<A.i; return A.u[i]
        elseif j==A.i&&i>A.i; return A.u[i-1]
        else
            return zero(A.D[1])
        # return zeros(size(A.D[i<A.i ? i : i-1],1),size(A.D[j<A.i ? j : j-1],1))
        end
    end

    Matrix(A::Arrow) =[A[i,j] for i=1:size(A,1), j=1:size(A,2)]
    adjoint(A::Arrow)=Arrow(adjoint.(A.D),A.v,A.u, adjoint(A.α),A.i)
    transpose(A::Arrow)=Arrow(A.D, conj.(A.u), conj.(A.v),A.α,A.i)

    # DPRk
    size(A::DPRk, dim::Integer) = length(A.Δ)
    size(A::DPRk)= size(A,1), size(A,1)

    function getindex(A::DPRk,i::Integer,j::Integer)
        # This is because Julia extracts rows as column vectors
        Aij=conj.(A.x[i,:])⋅(A.ρ*conj.(A.y[j,:]))
        return i==j ? A.Δ[i].+Aij : Aij
    end

    Matrix(A::DPRk)=[A[i,j] for i=1:size(A,1), j=1:size(A,2)]
    adjoint(A::DPRk)=DPRk(adjoint.(A.Δ),A.y,A.x,adjoint(A.ρ))
end
```

*()

- (generic function with 221 methods)

```
begin
    import Base:*,-
    function *(A::Arrow,z::Vector)
        n=size(A,1)
        T=typeof(A.u[1])
        w=Vector{T}(undef,n)
        i=A.i
        zi=z[i]
        for j=1:i-1
            w[j]=A.D[j]*z[j]+A.u[j]*zi
        end
        ind=[1:i-1;i+1:n]
        w[i]=A.v⋅z[ind]+A.α*zi
        # w[i]=adjoint(A.v[1:i-1])*z[1:i-1]+A.α*zi+adjoint(A.v[i:n-1])*z[i+1:n]
        for j=A.i+1:n
            w[j]=A.u[j-1]*zi+A.D[j-1]*z[j]
        end
        return w
    end

    function *(A::DPRk,z::Vector)
        n=size(A,1)
        T=typeof(A.x[1])
        w=Vector{T}(undef,n)
        β=A.ρ*(adjoint(A.y)*z)
        return Diagonal(A.Δ)*z+A.x*β
    end

    -(A::Arrow,D::Diagonal)=Arrow(A.D-D.diag[1:end-1],A.u,A.v,A.α-D.diag[end],A.i)
    -(A::DPRk,D::Diagonal)=DPRk(A.Δ-D.diag,A.x,A.y,A.ρ)

end
```

inv()

inv (generic function with 39 methods)

```julia
begin
    import LinearAlgebra.inv
    function inv(A::Arrow)
        j=findfirst(iszero.(A.D))
        if j==nothing
            p=[1:A.i-1;length(A.D)+1;A.i:length(A.D)]
            Δ=inv.(A.D)
            x=Δ.* A.u
            push!(x,-one(x[1]))
            y=adjoint.(Δ) .* A.v
            push!(y,-one(y[1]))
            ρ=inv(A.α-adjoint(A.v)*(Δ .*A.u))
            push!(Δ,zero(Δ[1]))
            return DPRk(Δ[p],x[p],y[p],ρ)
        else
            n=length(A.D)
            ind=[1:j-1;j+1:n]
            D=A.D[ind]
            u=A.u[ind]
            v=A.v[ind]
            pₕ=collect(1:n)
            deleteat!(pₕ,n)
            iₕ= (j>=A.i) ? A.i : A.i-1
            insert!(pₕ,iₕ,n)

            # Little bit elaborate to acommodate blocks
            Dₕ=inv.(D)
            uₕ=-Dₕ .* u
            push!(uₕ,one(uₕ[1]))
            uₕ*=inv(A.u[j])

            vₕ=-adjoint.(Dₕ) .* v
            push!(vₕ,one(D[1]))
            vₕ*=inv(A.v[j])

            αₕ=adjoint(inv(A.v[j]))*(-A.α+adjoint(v)*(Dₕ .* u))*inv(A.u[j])

            push!(Dₕ,zero(D[1]))
            jₕ=(j<A.i) ? j : j+1
            return Arrow(Dₕ[pₕ],uₕ[pₕ],vₕ[pₕ],αₕ,jₕ)
        end
    end

    function inv(A::DPRk)
        j=findfirst(iszero.(A.Δ))
        n=length(A.Δ)
        if j==nothing
            Δₕ=inv.(A.Δ)
            xₕ=Δₕ .* A.x
            yₕ=adjoint.(Δₕ) .* A.y
            ρₕ=-A.ρ*inv(I+adjoint(A.y)*(Δₕ .* (A.x*A.ρ)))
            return DPRk(Δₕ,xₕ,yₕ,ρₕ)
        else
            ind=[1:j-1;j+1:n]
            Δ=inv.(A.Δ[ind])
            x=A.x[ind,:]
            y=A.y[ind,:]
            uₕ=(-Δ .* x)*inv(A.x[j])
            vₕ=(-adjoint.(Δ) .* y)*inv(A.y[j])
            αₕ=adjoint(inv(A.y[j]))*(inv(A.ρ)+adjoint(y)*(Δ .* x)) *inv(A.x[j])
            println(" inv else ")
```

```
            return Arrow(Δ,uₕ,vₕ,αₕ,j)
        end
    end
end
```

Power()

Power (generic function with 3 methods)

```
function Power(A::AbstractMatrix{T},standardform::Bool=true,tol::Real=1e-12) where
T<:Number
    # Right eigenvalue and eigenvector of a (quaternion) Arrow matrix
    x=normalize!(randn(T,size(A,1)))
    y=A*x
    ν=x⋅y
    steps=1
    while norm(y-x*ν)>tol && steps<3000
        normalize!(y)
        x=y
        y=A*x
        ν=x⋅y
        # println(ν)
        steps+=1
    end
    if standardform
        z=standardformx(ν)
        ν=inv(z)*ν*z
        y.*=z
    end
    println("Power ", steps)
    normalize!(y)
    ν, y
end
```

RQI() and MRQI()

RQI (generic function with 3 methods)

```
function RQI(A::AbstractMatrix{T},standardform::Bool=true,tol::Real=1e-12) where
T<:Number
    # Right eigenvalue and eigenvector of a (quaternion) Arrow matrix
    # using Rayleigh Quotient Iteration
    n=size(A,1)
    x=normalize!(ones(T,n))
    # Only real shifts
    ν=(x'*x)\(x'*(A*x))
    μ=real(ν)
    y=inv(A-μ*I(n))*x
    normalize!(y)
    steps=1
    while norm(A*y-y*ν)>tol && steps<3000
        x=y
        ν=(x'*x)\(x'*(A*x))
        μ=real(ν)
        y=inv(A-μ*I(n))*x
        normalize!(y)
        # println(ν)
        steps+=1
    end
    if standardform
        z=standardformx(ν)
        ν=inv(z)*ν*z
        y.*=z
    end
    println("RQI ",steps)
    normalize!(y)
    ν, y
end
```

MRQI (generic function with 3 methods)

```
function MRQI(A::AbstractMatrix{T},standardform::Bool=true,tol::Real=1e-12) where
T<:Number
    # Right eigenvalue and eigenvector of a (quaternion) Arrow matrix
    # using Modified Rayleigh Quotient Iteration
    n=size(A,1)
    x=normalize!(ones(T,n))
    # Only real shifts
    ν=(transpose(x)*x)\(transpose(x)*(A*x))
    μ=real(ν)
    y=inv(A-μ*I(n))*x
    normalize!(y)
    steps=1
    while norm(A*y-y*ν)>tol && steps<3000
        x=y
        ν=(transpose(x)*x)\(transpose(x)*(A*x))
        μ=real(ν)
        y=inv(A-μ*I(n))*x
        normalize!(y)
        # println(ν)
        steps+=1
    end
    if standardform
        z=standardformx(ν)
        ν=inv(z)*ν*z
        y.*=z
    end
    println("MRQI ",steps)
    normalize!(y)
    ν, y
end
```

eigvals()

```
eigvals (generic function with 21 methods)
```

```julia
begin
    import LinearAlgebra.eigvals
    function eigvals(A₀::Arrow{T}, standardform::Bool=true) where T<:Number
        # Power iteration and Wielandt deflation to compute eigenvalues of
        # quaternionic Arrow matrix
        A=A₀
        n=size(A,1)
        # Create vector for eigenvalues
        λ=Vector{T}(undef,n)
        # First eigenpair
        λ[1],x=Esolver(A)
        for i=2:n-1
            # Deflated matrix
            g=x[1]\A.u[1]
            w=A.u[2:end]-x[2:end-1]*g
            α=A.α-x[end]*g
            A=Arrow(A.D[2:end],w,A.v[2:end],α,length(w)+1)
            # Eigenpair
            λ[i],x=Esolver(A)
            # println(u[1],A.ρ*A.y'*u)
        end
        # Last eigenvalue
        ν=A.α-x[2]*(x[1]\A.u[1])
        z=[one(T)]
        if standardform
            z=standardformx(ν)
            ν=inv(z)*ν*z
        end
        λ[n]=ν
        return λ
    end

    function eigvals(A₀::DPRk{T}, standardform::Bool=true) where T<:Number
        # Power iteration and Wielandt deflation to compute eigenvalues of
        # quaternionic DPR1 matrix
        A=A₀
        n=size(A,1)
        # Create vector for eigenvalues
        λ=Vector{T}(undef,n)
        # First eigenpair
        λ[1],u=Esolver(A)
        for i=2:n
            # Deflated matrix
            g=Matrix(transpose(u[1]\A.x[1,:]))
            x=A.x[2:end,:]-u[2:end]*g
            A=DPRk(A.Δ[2:end],x,A.y[2:end,:],A.ρ)
            # Eigenpair
            λ[i],u=Esolver(A)
            # println(u[1],A.ρ*A.y'*u)
        end
        return λ
    end
end
```

eigvecs()

eigvecs (generic function with 15 methods)

```
begin
    import LinearAlgebra.eigvecs
    function eigvecs(A::Arrow{T}, λ₁::Vector{T}, ψ₁::Vector{T}) where T<:Number
        # Eigenvectors of a (quaternionic) Arrow given eigenvalues λ₁ and last
        # elements ψ₁
        n=length(λ₁)
        # Create matrix for eigenvectors
        U=Matrix{T}(undef,n,n)
        # Temporary vector
        u=Vector{T}(undef,n)
        for i=1:n
            # Compute α=ρ*y'*u from the first element
            ψ=ψ₁[i]
            λ=λ₁[i]
            u[n]=ψ
            for k=1:n-1
                u[k]=sylvester(A.D[k],-λ,A.u[k]*ψ)
            end
            U[:,i]=u
        end
        return U
    end

    function eigvecs(A::DPRk{T}, λ₁::Vector{T}, ζ₁::Vector{Vector{T}}) where T<:Number
        # Eigenvectors of a (quaternionic) DPRk given eigenvalues λ₁ and first k
        # elements ζ₁, it should be n>k
        n=size(A,1)
        m=length(λ₁)
        k=size(A.x,2)
        # Create matrix for eigenvectors
        U=Matrix{T}(undef,n,m)
        # Temporary vector
        u=Vector{T}(undef,n)
        for i=1:m
            # Compute α=ρ*y'*u from the first k elements
            ζ=ζ₁[i]
            λ=λ₁[i]
            # α=-pinv(transpose(A.x[1,:]))*(A.Δ[1]*ζ-ζ*λ)
            # println(α," ", A.ρ*adjoint(A.y)*F.vectors[:,i])
            # α=A.ρ*adjoint(A.y)*F.vectors[:,i]
            α=A.x[1:k,:]\(ζ*λ-A.Δ[1:k].*ζ)
            u[1:k]=ζ
            for l=k+1:n
                u[l]=sylvester(A.Δ[l],-λ,transpose(A.x[l,:])*α)
            end
            normalize!(u)
            U[:,i]=u
        end
        U
    end
end
```

eigen()

```
eigen (generic function with 22 methods)
```

```julia
begin
    import LinearAlgebra.eigen
    function eigen(A₀::Arrow{T}, standardform::Bool=true) where T<:Number
        # Power iteration and Wielandt deflation to compute eigenvalues of
        # quaternionic Arrow matrix
        A=A₀
        n=size(A,1)
        # Create arrays for eigenvalues, first element and eigenvectors
        λ=Vector{T}(undef,n)
        γ=Vector{T}(undef,n)
        # First element of A.x
        χ=Vector{T}(undef,n)
        # x₁=Vector{T}(undef,n)
        # First and last elements of current u
        ν=Vector{T}(undef,n)
        ψ=Vector{T}(undef,n)
        # Eigenvector matrix
        U=zeros(T,n,n)

        # First eigenvalue
        λ[1],u=Esolver(A)
        γ[1]=u[1]*λ[1]/u[1]
        # U[:,1]=u
        ν[1]=u[1]
        χ[1]=A.u[1]
        ψ[1]=u[n]
        for i=2:n-1
            # Deflated matrix
            g=u[1]\A.u[1]
            w=A.u[2:end]-u[2:end-1]*g
            α=A.α-u[end]*g
            A=Arrow(A.D[2:end],w,A.v[2:end],α,length(w)+1)
            # Eigenpair
            λ[i],u=Esolver(A)
            γ[i]=u[1]*λ[i]/u[1]
            ν[i]=u[1]
            χ[i]=A.u[1]
            ψ[i]=u[end]
            # println(u[1],A.ρ*A.v'*u)
        end
        # Last eigenvalue
        μ=A.α-u[2]*(u[1]\A.u[1])
        z=[one(T)]
        if standardform
            z=standardformx(μ)
            μ=inv(z)*μ*z
        end
        λ[n]=μ
        ψ[n]=z

        # Compute the eigenvectors, bottom-up, the formulas are derived
        # using (4) and known first and last elements of eigenvectors
        for i=n-1:-1:1
            for j=i+1:n
                ζ=sylvester(γ[i],-λ[j],χ[i]*ψ[j])
                ν[j]=ζ
                ψ[j]=ψ[j]+ψ[i]*(ν[i]\ ζ)
            end
        end

        U=eigvecs(A₀,λ,ψ)
```

```
        return Eigen(λ,U)
    end

    function eigen(A₀::DPRk{T}, standardform::Bool=true) where T<:Number
        # Power iteration and Wielandt deflation to compute eigenvalues of
        # quaternionic DPR1 matrix
        A=A₀
        n=size(A,1)
        k=size(A.x,2)
        # Create arrays for eigenvalues, first elements and eigenvectors
        λ=Vector{T}(undef,n)
        γ=Vector{T}(undef,n)
        # First and second elements of A.x
        χ=Vector{Vector{T}}(undef,n)
        x₁=Vector{Matrix{T}}(undef,n)
        # First and second elements of current u
        ν=Vector{T}(undef,n)
        u₁=Vector{Vector{T}}(undef,n)
        # Eigenvector matrix
        U=zeros(T,n,n)

        # First eigenvalue
        λ[1],u=Esolver(A)
        γ[1]=u[1]*λ[1]/u[1]
        # Save elements of computed eigenvector
        # U[:,1]=u
        ν[1]=u[1]
        u₁[1]=u[2:k+1]
        χ[1]=A.x[1,:]
        x₁[1]=A.x[2:k+1,:]

        # Wielandt's deflation
        for i=2:n
            # Deflated matrix
            g=Matrix(transpose(u[1]\χ[i-1]))
            x=A.x[2:end,:]-u[2:end]*g
            A=DPRk(A.Δ[2:end],x,A.y[2:end,:],A.ρ)
            # Eigenpair of the deflated matrix
            λ[i],u=Esolver(A)
            γ[i]=u[1]*λ[i]/u[1]
            ν[i]=u[1]
            χ[i]=A.x[1,:]

            κ=min(k+1,length(u))
            u₁[i]=u[2:κ]
            x₁[i]=A.x[2:κ,:]
        end

        # Compute the eigenvectors, bottom-up, the formulas are derived
        # using (14) and known first elements
        for i=n-1:-1:1
            for j=i+1:n
                if length(u₁[j])==k
                    # Standard case
                    υ=[ν[j];(u₁[j])[1:k-1]]
                    α=(x₁[i]-u₁[i]*(1/ν[i])*transpose(χ[i]))\(υ*λ[j]-A₀.Δ[i+1:i+k].*υ)
                    ζ=sylvester(γ[i],-λ[j],transpose(χ[i])*α)
                    u₁[j]=υ.+u₁[i]*(1/ν[i])*ζ
                    ν[j]=ζ
                else
                    # Short case - direct formula
                    υ=u₁[j]
```

```
                        α=A.ρ*adjoint(A₀.y[n-length(υ):n,:])*[ν[j];υ]
                        ζ=sylvester(γ[i],-λ[j],transpose(χ[i])*α)
                        u₁[j]=[ν[j];υ[1:end]].+u₁[i]*(1/ν[i])*ζ
                        ν[j]=ζ
                    end
                end
            end

            ξ=Vector{Vector{T}}(undef,n)
            [ξ[i]=[ν[i];u₁[i][1:k-1]] for i=1:n]
            U=eigvecs(A₀,λ,ξ)
            return Eigen(λ,U)
        end
```

Examples

MRQI (generic function with 3 methods)

```
begin
    T=QuaternionF64
    # T=ComplexF64
    n=8
    Esolver=MRQI
    # Esolver=Power
    # Esolver=RQI
end
```

(6.1385e-12, 2.1802e-10)

```
ErrorA, ErrorB
```

```
begin
    # Arrow
    Random.seed!(5419)
    D₀=randn(T,n-1)
    u₀=randn(T,n-1)
    v₀=randn(T,n-1)
    α₀=randn(T)
    A=Arrow(D₀,u₀,v₀,α₀,n)
    if T==Float64
        # Treat everything as complex
        A=Arrow(ComplexF64.(A.D),ComplexF64.(A.u),ComplexF64.
        (A.v),ComplexF64(A.α),A.i)
    end

    # DPRk
    Random.seed!(5477)
    k=3
    Δ₀=randn(T,n)
    x₀=randn(T,n,k)
    y₀=randn(T,n,k)
    ρ₀=randn(T,k,k)
    B=DPRk(Δ₀,x₀,y₀,ρ₀)
    if T==Float64
        # Treat everything as complex
        B=DPRk(map.(ComplexF64,(B.Δ,B.x,B.y,B.ρ))...)
    end
end
```

```
8×8 Matrix{Quaternions.QuaternionF64}:
 QuaternionF64(-0.495836, 0.127426, 0.671807, 0.732388)  …   QuaternionF64(0.209486, -0.15
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)       QuaternionF64(-0.348121, 0.2
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)      QuaternionF64(-0.078416, 0.746
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)       QuaternionF64(0.939264, -0.09
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)       QuaternionF64(0.624517, 0.225
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)  …    QuaternionF64(0.509626, 0.39
                     QuaternionF64(0.0, 0.0, 0.0, 0.0)        QuaternionF64(-0.179395, 0.
  QuaternionF64(0.58344, 0.188744, 0.459965, -0.280137)      QuaternionF64(-0.433896, -0.3
```

```
1 Matrix(A)
```

```
8×8 Matrix{Quaternions.QuaternionF64}:
  QuaternionF64(-0.515588, 0.297013, 4.38071, 0.13383)  …     QuaternionF64(1.10436, -0.50
 QuaternionF64(0.848858, -1.97468, -4.10723, 0.473837)      QuaternionF64(-0.647778, 0.09
 QuaternionF64(-2.50889, -3.47454, -1.21862, 0.758868)        QuaternionF64(0.215771, -1.
  QuaternionF64(-2.70949, -2.58537, 4.06518, 0.490722)       QuaternionF64(0.682835, 0.36
  QuaternionF64(-1.92215, -2.12365, 3.28629, -1.84191)       QuaternionF64(-1.28105, 0.70
    QuaternionF64(3.23498, -2.53596, 2.20738, 3.52801)  …     QuaternionF64(0.677999, -1.
    QuaternionF64(1.02079, -0.6432, -0.821149, 2.2432)    QuaternionF64(-0.184563, -0.299
   QuaternionF64(-1.16923, -2.38838, 3.17517, 3.86446)        QuaternionF64(-0.30525, -0.
```

```
1 Matrix(B)
```

```
MRQI (generic function with 3 methods)
```

```
1 Esolver
```

```
(Quaternion(0.267802, 0.309375, 1.38778e-17, -1.38778e-17), [Quaternion(0.002989, -0.0900
```

```
1 ll,yy=Esolver(A)
```

```
MRQI 30
```

```
3.211931362804952e-13
```

```
1 norm(A*yy-yy*ll)
```

```
[Quaternion(0.267802, 0.309375, 1.38778e-17, -1.38778e-17), Quaternion(0.865239, 0.642197
```

```
1 eigvals(A)
```

```
MRQI 30
MRQI 41
MRQI 157
MRQI 119
MRQI 319
MRQI 185
MRQI 50
```

```
[-1.56879+1.37402im, -1.56879-1.37402im, -0.701061-0.99473im, -0.701061+0.99473im, -0.583
```

```
1 # Check
2 eigvals(unblock(q2c.(Matrix(A))))
```

```
E =
Eigen{Quaternions.QuaternionF64, Quaternions.QuaternionF64, Matrix{Quaternions.QuaternionF
values:
8-element Vector{Quaternions.QuaternionF64}:
 Quaternions.QuaternionF64(0.2678020444013188, 0.3093747800156201, 1.3877787807814457e-17,
  Quaternions.QuaternionF64(0.8652390085866449, 0.6421966139917064, -6.938893903907228e-17
 Quaternions.QuaternionF64(-0.5831962532417099, 0.6665047572335332, -5.551115123125783e-17
 Quaternions.QuaternionF64(-0.20331208025586178, 0.6990009404138914, 2.7755575615628914e-1
                     Quaternions.QuaternionF64(-0.7010613053528453, 0.9947298225412129, 0.
                   Quaternions.QuaternionF64(-0.05860520572131402, 0.8824196063854012, -6.
 Quaternions.QuaternionF64(-0.2873094856151356, 1.0040927077077575, -5.551115123125783e-17
  Quaternions.QuaternionF64(-1.5687909190781137, 1.374024603266676, 1.1102230246251565e-16
vectors:
8×8 Matrix{Quaternions.QuaternionF64}:
   QuaternionF64(0.002989, -0.0900966, -0.180056, -0.0992197)  …     QuaternionF64(-0.343
  QuaternionF64(-0.0788099, 0.0541528, 0.0531992, -0.0216652)      QuaternionF64(0.168583
    QuaternionF64(-0.072087, -0.171039, 0.0610917, -0.136404)         QuaternionF64(-0.08
   QuaternionF64(0.00824405, 0.101283, -0.153285, -0.0518963)     QuaternionF64(-0.049131
 QuaternionF64(0.0546324, -0.0407192, -0.0669173, -0.0373658)        QuaternionF64(-0.1884
     QuaternionF64(-0.653135, 0.0345469, 0.150565, -0.193848)  …  QuaternionF64(0.042951,
       QuaternionF64(-0.202392, 0.46517, -0.171245, 0.175308)    QuaternionF64(0.267748,
   QuaternionF64(0.077213, -0.0136997, -0.14745, -0.00496785)          QuaternionF64(0.2
```

```
E=eigen(A)
```

```
MRQI 30
MRQI 41
MRQI 157
MRQI 119
MRQI 319
MRQI 185
MRQI 50
```

```
ErrorA = 6.1384954634723205e-12
```

```
ErrorA=norm(Matrix(A)*E.vectors-E.vectors*Diagonal(E.values))
```

```
(Quaternion(0.318951, 0.290079, -6.93889e-18, 6.93889e-18), [Quaternion(-0.0144122, 0.006
```

```
Esolver(B)
```

```
MRQI 30
```

```
[Quaternion(0.318951, 0.290079, -6.93889e-18, 6.93889e-18), Quaternion(0.663183, 0.694753
```

```
eigvals(B)
```

```
MRQI 30
MRQI 96
MRQI 41
MRQI 35
MRQI 64
MRQI 25
MRQI 48
MRQI 1
```

```
[-4.86236+5.07097im, -4.86236-5.07097im, -0.841801+0.47291im, -0.841801-0.47291im, -0.608
```

```
# Check
eigvals(unblock(q2c.(Matrix(B))))
```

```
F =
Eigen{Quaternions.QuaternionF64, Quaternions.QuaternionF64, Matrix{Quaternions.QuaternionF
values:
8-element Vector{Quaternions.QuaternionF64}:
   Quaternions.QuaternionF64(0.3189514252247643, 0.2900790447694579, -6.938893903907228e-1
                     Quaternions.QuaternionF64(0.6631833605531319, 0.6947530758709467, 0.0
  Quaternions.QuaternionF64(-0.8418005583369264, 0.47291002224072315, 6.938893903907228e-1
                    Quaternions.QuaternionF64(-0.1081668286493929, 0.6566250305203121, 0.0
  Quaternions.QuaternionF64(-0.6080414371475934, 1.651785373144668, -8.326672684688674e-17
 Quaternions.QuaternionF64(1.1627106891762002, 2.0068176646206335, 2.7755575615628914e-17,
     Quaternions.QuaternionF64(-4.862363023625785, 5.070973270608173, 4.440892098500626e-1
    Quaternions.QuaternionF64(1.7705524659918832, 7.395067910968743, 6.661338147750939e-16
vectors:
8×8 Matrix{Quaternions.QuaternionF64}:
 QuaternionF64(-0.0144122, 0.00633799, -0.0114835, 0.0334537)  …   QuaternionF64(-0.240531
  QuaternionF64(0.0383775, 0.0225131, 0.00459511, -0.0231533)        QuaternionF64(0.12888
   QuaternionF64(-0.0603084, -0.0569127, 0.179312, 0.0140731)      QuaternionF64(0.011632,
     QuaternionF64(0.015655, -0.0193163, 0.220507, 0.0670487)       QuaternionF64(-0.309735
     QuaternionF64(-0.043433, 0.0194125, 0.0943529, 0.116561)         QuaternionF64(0.11085
     QuaternionF64(-0.0475708, 0.307915, 0.345969, -0.193899)  …    QuaternionF64(0.0106646
  QuaternionF64(-0.0375769, -0.714434, -0.166422, -0.0309928)       QuaternionF64(0.14365,
     QuaternionF64(0.0790173, -0.0424579, 0.162597, 0.221162)        QuaternionF64(0.13091
```

```
F=eigen(B)
```

```
MRQI 30
MRQI 96
MRQI 41
MRQI 35
MRQI 64
MRQI 25
MRQI 48
MRQI 1
```

```
ErrorB = 2.1802002523641636e-10
```

```
ErrorB=norm(Matrix(B)*F.vectors-F.vectors*Diagonal(F.values))
```