GlobalMaxEigenpair Vignette

Mu-Fa Chen
Assisted by Xiao-Jun Mao

2017-10-01

`GlobalMaxEigenpair` is a continuation package of `EfficientMaxEigenpair` for computing the maximal eigenpair of the matrices with non-negative off-diagonal elements (or as a dual, the minimal eigenvalue of the matrices with non-positive off-diagonal elements). It introduces mainly two global algorithms for computing the maximal eigenpair in a rather general setup, including even a class of real (with some negative off-diagonal elements) or complex matrices. This vignette is a simple guide to using the package. All the algorithms and examples provided in this vignette are available in the paper "Global algorithms for maximal eigenpair" by Mu-Fa Chen. The paper Chen (2017) is now included in the Vol 4, in the middle of the website:

http://math0.bnu.edu.cn/~chenmf/

For the comparison, let us install and require both the packages `EfficientMaxEigenpair` and `GlobalMaxEigenpair` first.

```
require(EfficientMaxEigenpair)
require(GlobalMaxEigenpair)
```

We now introduce the package in details. It offers two main algorithms:

- `global.maxeig.general()`: Calculate the maximal eigenpair for the general matrix by using global algorithms. Let the general matrix $A = (a_{ij} : i, j \in E)$.
  - There are two choices for the option `complex`. `complex=T` if the input matrix A is a complex matrix and `complex=F` if the input matrix A is a real matrix. The default one is `complex=F`. See Example 15 in the following for illustration.
  - There are two choices of the global algorithms. The default one `alg="ray.quot"` corresponding to the Rayleigh quotient iteration. The option `alg="shifted.inv"` corresponding to the shifted inverse iteration. The difference between `ray.quot()` and `shifted.inv()` is that we shift the convergence sequences $y^{(n)} = \max_{0\le j\le N} \frac{(Aw^{(n)})_j}{w_j^{(n)}}$ and $z^{(n)} = v^{(n)*}Av^{(n)}$ to be $y^{(n)} = v^{(n)*}Av^{(n)}$ and $z^{(n)} = \max_{0\le j\le N} \frac{(Aw^{(n)})_j}{w_j^{(n)}}$. Finally, we output the approximation by using the same notation $(z^{(n)}, v^{(n)})$. In general, `ray.quot()` is often a little effective than `shifted.inv()`, saving one iteration for instance, but in `shifted.inv()`, each iteration is safe, never failed into a pitfall.
  - The initial vector w_0 is given by the option `w0`. Typically, we use the uniformly distributed one as the initial vectors in all the global algorithms.
  - There are three choices of the initial z_0 as the approximation of $\rho(A)$ which is the maximal eigenvalue of the general matrix A. The option `z_0="fixed"` corresponding to the default one, i.e, set $z_0 = \max_{0\le i\le N}(A * w_0)_i$. The option `z_0="convex"` corresponding to a convex combination, i.e, set $z_0 = \xi v_0^* A v_0 + (1-\xi)\min_{0\le i\le N}(A * w_0)_i$. The option `z_0="numeric"` corresponding to some particular choices, we directly set the value of z_0 based on past experience or tested.
  - The option `z0numeric` is only used when the particular initial z_0 is considered.
  - The precise level used for output results is set to be `digit.thresh = 6` which implies 1e-6 without any special requirement. Same for the following three algorithms and the examples shown in this vignette.
  - The coefficient `xi` is used in the improved initial z_0 to form the convex combination of $v_0^* A v_0$ and $\min_{0\le i\le N}(A * w_0)_i$ (in the dual case $v_0^*(-Q)v_0$ and $\max_{0\le i\le N}(-Q * w_0)_i$), it should between 0 and 1.
- `global.maxeig.non()`: Calculate the maximal eigenpair for the general matrix with non-negative off-diagonal elements by using global algorithm with Rayleigh quotient iteration. Let the general matrix

$Q = (q_{ij} : i, j \in E)$. Instead of studying the maximal eigenpair of Q, we study the minimal eigenpair of $-Q$. This algorithm shares lots similarity with `global.maxeig.general()` by replacing the matrix A with matrix $-Q$.

  - There are three choices of the global algorithms. The default one `alg="ray.quot.non"` corresponding to the Rayleigh quotient iteration. The option `alg="shifted.inv.non"` corresponding to the shifted inverse iteration and `alg="mod.ray.quot.non"` corresponding to Algorithm 4_2 in Chen (2016). The difference between `ray.quot.non()` and `shifted.inv.non()` is that we shift the convergence sequences $x^{(n)} = \min_{0\le j\le N} \frac{((-Q)w^{(n)})_j}{w_j^{(n)}}$ and $z^{(n)} = v^{(n)*}(-Q)v^{(n)}$ to be $x^{(n)} = v^{(n)*}(-Q)v^{(n)}$ and $z^{(n)} = \min_{0\le j\le N} \frac{((-Q)w^{(n)})_j}{w_j^{(n)}}$. For `mod.ray.quot.non()`, for $z^{(k)} : k = 0, 1, 2$, we choose $z^{(k)}$ to be the same as those defined in `shifted.inv.non()` and for $k \ge 3$, keep $z^{(k)}$ to be the same as those defined in `ray.quot.non()`.
  - For the option `z_0="fixed"`, we set $z_0 = 0$. The option `z_0="convex_1"` corresponding to one convex combination by setting $z_0 = \xi v_0^*(-Q)v_0$. The option `z_0="convex_2"` corresponding to another convex combination by set $z_0 = \xi v_0^*(-Q)v_0 + (1-\xi)\max_{0\le i\le N}(-Q * w_0)_i$.
  - Similarly, there are some common options `w0`, `xi` and `digit.thresh` which are the same as defined in function `global.maxeig.general()`.

In summary, there five auxiliary functions `ray.quot()`, `ray.quot.non()`, `mod.ray.quot.non()`, `shifted.inv()` and `shifted.inv.non()` where:

- `ray.quot()`: Rayleigh quotient iteration algorithm for computing the maximal eigenpair of general matrix A.
- `ray.quot.non()`: Rayleigh quotient iteration algorithm for computing the maximal eigenpair of matrix with non-negative off-diagonal elements Q.
- `mod.ray.quot.non()`: Modified rayleigh quotient iteration algorithm for computing the maximal eigenpair of matrix with non-negative off-diagonal elements Q.
- `shifted.inv()`: Shifted inverse iteration algorithm for computing the maximal eigenpair of general matrix A.
- `shifted.inv.non()`: Shifted inverse iteration algorithm for computing the maximal eigenpair of matrix with non-negative off-diagonal elements Q.

Besides, we add a particular one `branch()` for the special use for branching process where:

- `branch()`: Generate a series of general matrix A motivated by the classical branching process with parameters α and N. The matrices used in Examples 8 and 9 are generated by this function.

Acknowledgement

The research project is supported in part by the National Natural Science Foundation of China (No. 11131003 and 11626245), the "985" project from the Ministry of Education in China, and the Project Funded by the Priority Academic Program Development of Jiangsu Higher Education Institutions. The first author acknowledges the assisted one for his hard work in preparing this package in R.

Examples

We show most of the examples in the paper Chen (2017) in this section. For a convenient comparison, we keep the same subsection names and examples as the paper Chen (2017). Armed with `global.maxeig.general()` and `global.maxeig.non()` we can calculate the maximal eigenpair for the general matrix and extend to the matrix with non-negative off-diagonal elements. Sometimes we will report the minimal eigenvalue of $-Q$ by the convention in the paper Chen (2017).

The minimal eigenvalue of -Q Example 6 (Same as Example 21 in Chen (2016))

1. The outputs by Algorithm 1 in the paper Chen (2017).

```
b4 = c(0.01, 1, 100, 10^4)
digits = c(9, 7, 6, 6)

for (i in 1:4) {
    A = matrix(c(-3, 4, 0, 10, 0, 2, -7, 5, 0, 0, 0, 3, -5, 0, 0, 1, 0, 0, -16,
        11, 0, 0, 0, 6, -11 - b4[i]), 5, 5)
    print(-global.maxeig.general(A, alg = "ray.quot", w0 = rep(1, dim(A)[1]),
        z0 = "fixed", digit.thresh = digits[i])$z[-1])
}
```

```
## [1] 0.000278773 0.000278686
## [1] 0.0251531 0.0245175
## [1] 0.191729 0.182822 0.182819
## [1] 0.201695 0.195019 0.195015
```

2. The outputs by Algorithm 2 in the paper Chen (2017).

```
for (i in 1:4) {
    A = matrix(c(-3, 4, 0, 10, 0, 2, -7, 5, 0, 0, 0, 3, -5, 0, 0, 1, 0, 0, -16,
        11, 0, 0, 0, 6, -11 - b4[i]), 5, 5)
    print(-global.maxeig.general(A, alg = "shifted.inv", w0 = rep(1, dim(A)[1]),
        z0 = "fixed", digit.thresh = digits[i])$z[-1])
}
```

```
## [1] 0.000278637 0.000278686
## [1] 0.0241546 0.0245175
## [1] 0.168776 0.182750 0.182819
## [1] 0.179525 0.194932 0.195015
```

3. The outputs by the algorithm in the paper Chen (2016).

```
for (i in 1:3) {
    A = matrix(c(-3, 4, 0, 10, 0, 2, -7, 5, 0, 0, 0, 3, -5, 0, 0, 1, 0, 0, -16,
        11, 0, 0, 0, 6, -11 - b4[i]), 5, 5)
    print(-eff.ini.maxeig.general(A, z0 = "Auto", digit.thresh = digits[i])$z[-1])
}
```

```
## [1] 0.000278151 0.000278686
## [1] 0.0236258 0.0245174 0.0245175
## [1] 0.200058 0.182609 0.182819
```

These tables show that the three algorithms are more or less at the same level of effectiveness. However, the first two are actually more economic since the last one requires an extra work computing the initial v_0.

Example 7 (So called single birth Q-matrix)

1. The outputs for different N by Algorithm 5 in the paper Chen (2017).

```
nn = c(8, 16, 32, 50, 100, 500, 1000, 5000, 10^4) - 1
for (i in 1:9) {
    Q = matrix(0, nn[i] + 1, nn[i] + 1)
    a = 1/seq(2, nn[i] + 1)
    b = c(1:nn[i])
    Q[, 1] = c(-1, a)
    diag(Q) = c(-1, -(a[1:(nn[i] - 1)] + b[2:nn[i]]), -a[nn[i]] - nn[i] - 1)
    Q[cbind(seq(1, nn[i]), seq(2, nn[i] + 1))] = b

    print(global.maxeig.non(Q, alg = "shifted.inv.non", w0 = rep(1, dim(Q)[1]),
        z0 = "fixed")$z[-1])
}
```

```
## [1] 0.276727 0.427307 0.451902 0.452339
## [1] 0.222132 0.367827 0.399959 0.400910
## [1] 0.187826 0.329646 0.370364 0.372308 0.372311
## [1] 0.171657 0.311197 0.357814 0.360776 0.360784
## [1] 0.152106 0.287996 0.343847 0.349166 0.349197
## [1] 0.121403 0.247450 0.321751 0.336811 0.337186
## [1] 0.111879 0.233257 0.313274 0.334155 0.335009 0.335010
## [1] 0.094743 0.205212 0.293025 0.328961 0.332609 0.332635
## [1] 0.088896 0.194859 0.284064 0.326285 0.332113 0.332188
```

2. The outputs for different N by the algorithm in the paper Chen (2016).

```
for (i in 1:9) {
    Q = matrix(0, nn[i] + 1, nn[i] + 1)
    a = 1/seq(2, nn[i] + 1)
    b = c(1:nn[i])

    Q[, 1] = c(-1, a)
    diag(Q) = c(-1, -(a[1:(nn[i] - 1)] + b[2:nn[i]]), -a[nn[i]] - nn[i] - 1)
    Q[cbind(seq(1, nn[i]), seq(2, nn[i] + 1))] = b

    print(-eff.ini.maxeig.general(Q, z0 = "Auto")$z[-1])
}
```

```
## [1] 0.450694 0.452338 0.452339
## [1] 0.39952 0.40091
## [1] 0.371433 0.372311
## [1] 0.360314 0.360784
## [1] 0.349501 0.349197
## [1] 0.340666 0.337185 0.337186
## [1] 0.340871 0.335003 0.335010
## [1] 0.347505 0.332536 0.332635
## [1] 0.352643 0.331975 0.332188
```

Clearly, the general algorithm introduced in Chen (2016) is efficient for this nonsymmetrizable model. We have seen that the present algorithms require more iterations than the earlier one, this is reasonable since the computations of the initials are excluded from the last table.

The numbers of iterations of Algorithms 4 and 4_2 in Chen (2017).

The first row represents the dimension of matrix.

```
nn = c(8, 16, 32, 50, 100, 500, 1000, 5000, 10^4) - 1

itermat = matrix(0, 2, 9)
colnames(itermat) = nn + 1
```

```r
rownames(itermat) = c("Algorithm 4", "Algorithm 4_2")

for (i in 1:9) {

    Q = matrix(0, nn[i] + 1, nn[i] + 1)
    a = 1/seq(2, nn[i] + 1)
    b = c(1:nn[i])

    Q[, 1] = c(-1, a)
    diag(Q) = c(-1, -(a[1:(nn[i] - 1)] + b[2:nn[i]]), -a[nn[i]] - nn[i] - 1)
    Q[cbind(seq(1, nn[i]), seq(2, nn[i] + 1))] = b

    global.maxeig.ray.quot.non = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "fixed")

    itermat[1, i] = global.maxeig.ray.quot.non$iter

    if (i > 4) {
        global.maxeig.mod.ray.quot.non = global.maxeig.non(Q, alg = "mod.ray.quot.non",
            w0 = rep(1, dim(Q)[1]), z0 = "fixed")

        itermat[2, i - 4] = global.maxeig.mod.ray.quot.non$iter

    }

}

itermat[2, 1:4] = c(" ", " ", " ", " ")

as.table(itermat)
```

```
##               8 16 32 50 100 500 1000 5000 10000
## Algorithm 4   3 3  3  4  4   4   5    6    7
## Algorithm 4_2            4   5   5    5    5
```

The numbers of iterations of Algorithms 4 and 5 with convex combination in Chen (2017).

Because $\lambda_{\min}(-Q) \in (0, v_0^*(-Q)v_0)$, we choose the convex combination $z_0 = \xi v_0^*(-Q)v_0 + (1-\xi) * 0 = \xi v_0^*(-Q)v_0$ for some $\xi \in (0,1)$ here. Namely, we set the option `z_0="convex_1"` in the following. The first row represents the dimension of matrix.

```r
nn = c(8, 16, 32, 50, 100, 500, 1000, 5000, 10^4) - 1

itermat = matrix(0, 5, 9)
colnames(itermat) = nn + 1
rownames(itermat) = c("Algorithm 4, xi = 0.34189", "Algorithm 4, xi = 0.23",
    "Algorithm 4, xi = 0.3", "Algorithm 5, xi = 0.23", "Algorithm 5, xi = 0.3")

for (i in 1:9) {

    Q = matrix(0, nn[i] + 1, nn[i] + 1)
    a = 1/seq(2, nn[i] + 1)
    b = c(1:nn[i])
    Q[, 1] = c(-1, a)
    diag(Q) = c(-1, -(a[1:(nn[i] - 1)] + b[2:nn[i]]), -a[nn[i]] - nn[i] - 1)
    Q[cbind(seq(1, nn[i]), seq(2, nn[i] + 1))] = b

    global.maxeig.conv.comb.ray.quot.non.1 = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_1", xi = 0.34189)
    itermat[1, i] = global.maxeig.conv.comb.ray.quot.non.1$iter

    global.maxeig.conv.comb.ray.quot.non.2 = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_1", xi = 0.23)
    itermat[2, i] = global.maxeig.conv.comb.ray.quot.non.2$iter

    global.maxeig.conv.comb.ray.quot.non.3 = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_1", xi = 0.3)
    itermat[3, i] = global.maxeig.conv.comb.ray.quot.non.3$iter

    global.maxeig.conv.comb.shifted.inv.non.1 = global.maxeig.non(Q, alg = "shifted.inv.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_1", xi = 0.23)
    itermat[4, i] = global.maxeig.conv.comb.shifted.inv.non.1$iter

    global.maxeig.conv.comb.shifted.inv.non.2 = global.maxeig.non(Q, alg = "shifted.inv.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_1", xi = 0.3)
    itermat[5, i] = global.maxeig.conv.comb.shifted.inv.non.2$iter
}

as.table(itermat)
```

```
##                              8 16 32 50 100 500 1000 5000 10000
## Algorithm 4, xi = 0.34189 3  3  3  3   3   3    3    3     3
## Algorithm 4, xi = 0.23    3  3  3  3   3   3    3    4     4
## Algorithm 4, xi = 0.3     3  3  3  3   3   3    3    3     4
## Algorithm 5, xi = 0.23    4  4  4  4   4   4    5    5     5
## Algorithm 5, xi = 0.3     4  4  4  4   4   4    4    4     4
```

From the above we can see that using the idea in the paper Chen (2017)… ($z_k{=}\delta_k^{-1}$) may need one or two more steps than the Chen (2016) but Chen (2017)'s improvement is safe and never fall into pitfall, so we can use the new one instead of the old in 2016.

Example 8 (Motivated from the classical branching process with $\alpha = 1$)

```
Q = branch(1, 8)
print(global.maxeig.non(Q, alg = "shifted.inv.non", w0 = rep(1, dim(Q)[1]),
    z0 = "fixed", digit.thresh = 7)$z[-1])

Q = branch(1, 16)
print(global.maxeig.non(Q, alg = "shifted.inv.non", w0 = rep(1, dim(Q)[1]),
    z0 = "fixed", digit.thresh = 8)$z[-1])
```

```
## [1] 0.0311491 0.0346044 0.0346310
## [1] 0.00256281 0.00260088
```

Example 9 (Motivated from the classical branching process with $\alpha = 7/4$)

The numbers of iterations of Algorithms 4 and 5 in Chen (2017).

The first row represents the dimension of matrix.

```
nn = c(8, 16, 50, 100, 500, 1000, 5000, 10^4)

itermat = matrix(0, 2, 8)
colnames(itermat) = nn
rownames(itermat) = c("Algorithm 4", "Algorithm 5")

for (i in 1:8) {

    Q = branch(7/4, nn[i])

    global.maxeig.ray.quot.non = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "fixed", digit.thresh = 6)
    itermat[1, i] = global.maxeig.ray.quot.non$iter

    global.maxeig.shifted.inv.non = global.maxeig.non(Q, alg = "shifted.inv.non",
        w0 = rep(1, dim(Q)[1]), z0 = "fixed", digit.thresh = 6)
    itermat[2, i] = global.maxeig.shifted.inv.non$iter

}

as.table(itermat)
```

```
##             8 16 50 100 500 1000 5000 10000
## Algorithm 4 5  6  7   7   8    8    9    10
## Algorithm 5 6  6  7   7   8    9    9    10
```

The numbers of iterations of Algorithms 4 and 5 with convex combination in Chen (2017).

Here we choose the convex combination to be $z_0 = \xi v_0^*(-Q)v_0 + (1-\xi)\max_{0\le i\le N}(-Q * w_0)_i$. Namely, we set the option `z_0="convex_2"` in the following. The first row represents the dimension of matrix.

```
nn = c(8, 16, 50, 100, 500, 1000, 5000, 10^4)

itermat = matrix(0, 2, 8)
colnames(itermat) = nn
rownames(itermat) = c("Algorithm 4", "Algorithm 5")

for (i in 1:8) {

    Q = branch(7/4, nn[i])

    global.maxeig.conv.comb.ray.quot.non = global.maxeig.non(Q, alg = "ray.quot.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_2", xi = 0.31, digit.thresh = 6)

    itermat[1, i] = global.maxeig.conv.comb.ray.quot.non$iter

    global.maxeig.conv.comb.shifted.inv.non = global.maxeig.non(Q, alg = "shifted.inv.non",
        w0 = rep(1, dim(Q)[1]), z0 = "convex_2", xi = 0.31, digit.thresh = 6)

    itermat[2, i] = global.maxeig.conv.comb.shifted.inv.non$iter
}
```

```r
as.table(itermat)
```

```
##              8 16 50 100 500 1000 5000 10000
## Algorithm 4 3  3  4   4   4    4    4     4
## Algorithm 5 3  3  4   4   4    4    4     4
```

The outputs of Algorithm 4 with convex combination in the subcritical case.

```r
nn = c(8, 16, 50, 100, 500, 1000, 5000, 10^4)

for (i in 1:8) {
    Q = branch(7/4, nn[i])

    print(global.maxeig.non(Q, alg = "shifted.inv.non", w0 = rep(1, dim(Q)[1]),
        z0 = "convex_2", xi = 0.31, digit.thresh = 6)$z[-1])
}
```

```
## [1] 0.637762 0.638152 0.638153
## [1] 0.621218 0.625480 0.625539
## [1] 0.609724 0.623933 0.624996 0.625000
## [1] 0.606792 0.623221 0.624989 0.625000
## [1] 0.604371 0.621944 0.624955 0.625000
## [1] 0.604062 0.621519 0.624935 0.625000
## [1] 0.603813 0.620682 0.624877 0.625000
## [1] 0.603782 0.620361 0.624846 0.625000
```

A class of real or complex matrices

This section shows that our algorithms work for some real and complex examples.

Example 12 (Real case)

```r
A = matrix(c(-1, 8, -1, 8, 8, 8, -1, 8, 8), 3, 3)

global.maxeig.general(A, alg = "ray.quot", w0 = rep(1, dim(A)[1]), z0 = "numeric",
    z0numeric = 24, digit.thresh = 4)$z[-1]

global.maxeig.general(A, alg = "shifted.inv", w0 = rep(1, dim(A)[1]), z0 = "numeric",
    z0numeric = 24, digit.thresh = 4)$z[-1]
```

```
## [1] 17.3772 17.5124
## [1] 18.5316 17.5416 17.5124
```

Example 15 (Complex case)

```r
A = matrix(c(0.75 - (0+1.125i), -0.5 - (0+1i), 2.75 - (0+0.125i), 0.5882 - (0+0.1471i),
    2.1765 + (0+0.7059i), 0.5882 - (0+0.1471i), 1.0735 + (0+1.4191i), 2.1471 -
        (0+0.4118i), -0.9265 + (0+0.4191i)), 3, 3)

global.maxeig.general(A, complex = T, alg = "shifted.inv", w0 = rep(1, dim(A)[1]),
    z0 = "fixed", digit.thresh = 5)$z[-1]
```

```
## [1] 3.03949-0.045160i 3.00471-0.001577i 2.99998-0.000027i 2.99997-0.000029i
```

References

```
## [1] M. F. Chen. "Efficient initials for computing maximal
## eigenpair". In: _Frontiers of Mathematics in China_ 11.6 (2016),
## pp. 1379-1418.
##
## [2] M. F. Chen. "Global algorithms for maximal eigenpair". In:
## _arXiv preprint arXiv:1706.07584_ (2017).
```