Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Basic Iterative Methods for Eigenvalue Problems

by Amir Bouslama and Nicolas Venkovic

Chair of Computational Mathematics
School of Computation, Information and Technology (CIT)
Technical University of Munich, Germany

[1]:
```
using LinearAlgebra, Plots, Printf, Latexify, LaTeXStrings, BenchmarkTools,␣
↪Random, SparseArrays, Random
```

1. *Create a matrix with prescribed eigenvalues.*

[2]:
```
function matrix_with_eigenvalues(prescribed_eigvals::AbstractVector)
  n = length(prescribed_eigvals)
  Λ = Diagonal(prescribed_eigvals)
  X = randn(n, n)
  A = X * Λ * inv(X)
  return A
end;
```

[3]:
```
n = 20
Random.seed!(1)
prescribed_eigvals = sort(rand(n)) .* 100
A = matrix_with_eigenvalues(prescribed_eigvals);
```

[4]:
```
computed_eigvals = eigvals(A);
for i in 1:n
  println("i = $i, prescribed_eigval = $(prescribed_eigvals[i]),␣
↪computed_eigval = $(computed_eigvals[i])")
end
```

```
i = 1, prescribed_eigval = 0.8197858843363659, computed_eigval =
0.8197858843364842
i = 2, prescribed_eigval = 2.4094310524527707, computed_eigval =
2.409431052453012
i = 3, prescribed_eigval = 4.91718221481211, computed_eigval = 4.917182214812215
i = 4, prescribed_eigval = 8.725304891274233, computed_eigval =
```

```
8.725304891274234
i = 5, prescribed_eigval = 11.907881640750706, computed_eigval =
11.907881640750706
i = 6, prescribed_eigval = 13.822702472322401, computed_eigval =
13.822702472322575
i = 7, prescribed_eigval = 19.852090963358837, computed_eigval =
19.85209096335874
i = 8, prescribed_eigval = 34.751338616076566, computed_eigval =
34.75133861607654
i = 9, prescribed_eigval = 34.77368477058109, computed_eigval =
34.77368477058095
i = 10, prescribed_eigval = 39.32710232252806, computed_eigval =
39.327102322528184
i = 11, prescribed_eigval = 59.20407556271403, computed_eigval =
59.20407556271369
i = 12, prescribed_eigval = 66.1425351684768, computed_eigval =
66.14253516847647
i = 13, prescribed_eigval = 69.18572875342215, computed_eigval =
69.18572875342225
i = 14, prescribed_eigval = 76.75180540873912, computed_eigval =
76.75180540873873
i = 15, prescribed_eigval = 77.81487141858243, computed_eigval =
77.81487141858233
i = 16, prescribed_eigval = 80.10551449181887, computed_eigval =
80.10551449181888
i = 17, prescribed_eigval = 80.25607099234905, computed_eigval =
80.25607099234898
i = 18, prescribed_eigval = 84.03841370820818, computed_eigval =
84.03841370820786
i = 19, prescribed_eigval = 85.57176841095733, computed_eigval =
85.57176841095735
i = 20, prescribed_eigval = 89.07696748195568, computed_eigval =
89.0769674819555
```

2. *Implement and test the power iteration. Evaluate the convergence rate.*

[5]:
```
function power_iteration(A::AbstractMatrix, q0::Vector{Float64}, max_iter::
↪Int=200, tol::Float64=1e-6)
 n = length(q0)
 q = normalize(q0)
 q_hist = Matrix{Float64}(undef, n, max_iter + 1)
 lbda_hist = zeros(Float64, max_iter)

 q_hist[:, 1] = q
 lbda_old = 0.0
 k = 0
 while k < max_iter
   k += 1
```

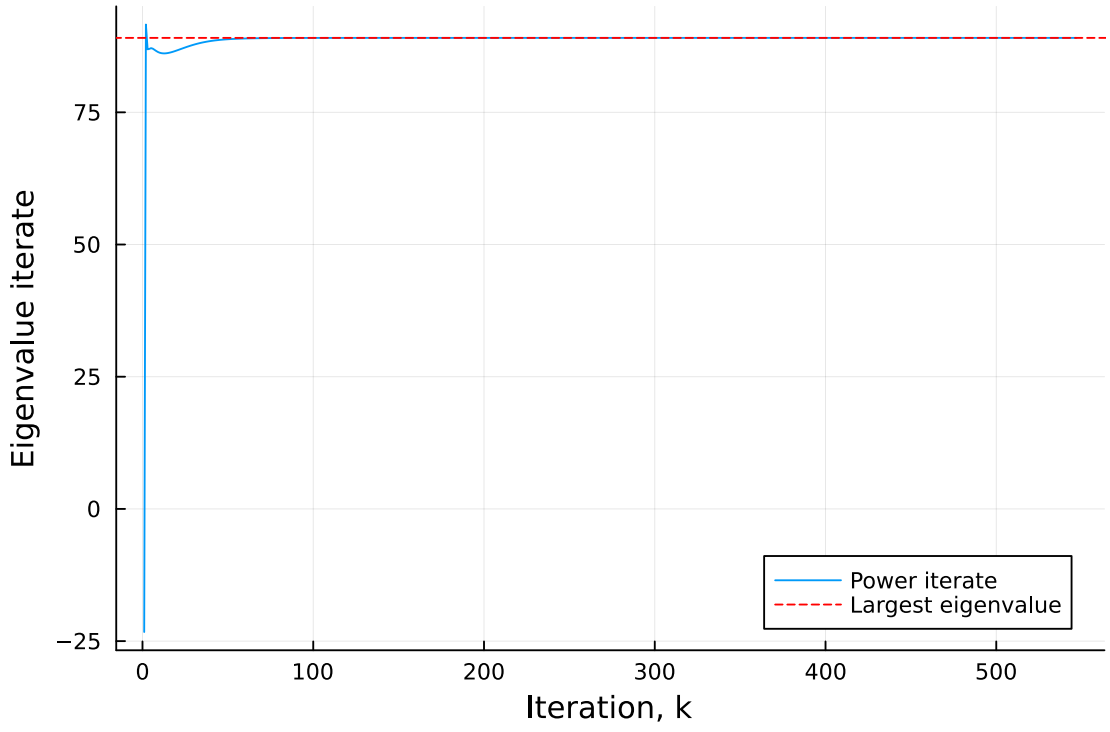
```
    z = A * q
    lbda = dot(q, z)
    lbda_hist[k] = lbda
    q = z / norm(z)
    q_hist[:, k + 1] = q
    if abs(lbda - lbda_old) < tol
      break
    end
    lbda_old = lbda
  end

  return lbda_hist[1:k], q_hist[:, 1:(k + 1)]
end;
```

[6]:
```
Random.seed!(1)
q0 = rand(n)
power_iterates, _ = power_iteration(A, q0, 1_000, 1e-12);
```

[7]:
```
lbda1 = prescribed_eigvals[end]
plot(power_iterates, label="Power iterate", xlabel="Iteration, k",␣
 ↪ylabel="Eigenvalue iterate")
hline!([lbda1], label="Largest eigenvalue", linestyle=:dash, color=:red)
```

[7]:

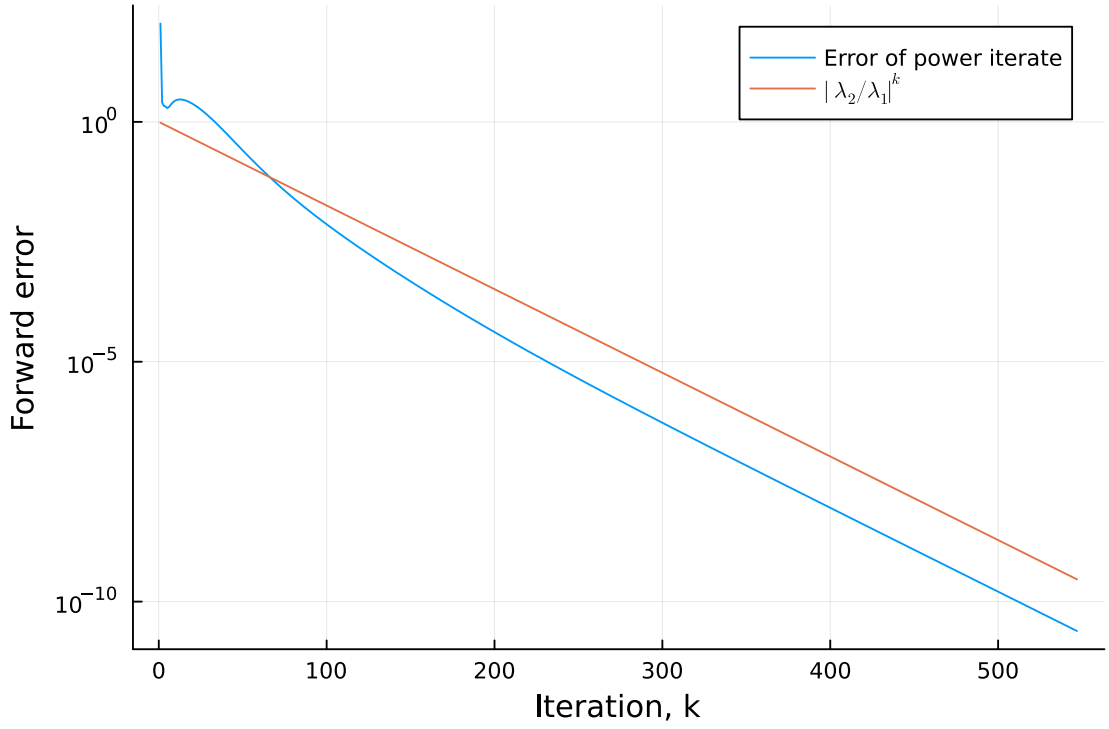


We know that the forward error of the power iterate converges in $O(|\lambda_2/\lambda_1|^k)$ where λ_1 and λ_2 are

the largest and second largest eigenvalues.

[8]:
```
lbda1 = prescribed_eigvals[end]
lbda2_over_lbda1 = prescribed_eigvals[end-1] / lbda1
power_errors = abs.(power_iterates .- lbda1)
plot(power_errors, yscale=:log10, xlabel="Iteration, k", ylabel="Forward␣
 ↪error", label="Error of power iterate")
plot!([lbda2_over_lbda1^k for k in 1:length(power_iterates)],␣
 ↪label=L"|\lambda_2/\lambda_1|^k")
```

[8]:



[9]:
```
for k in 1:length(power_iterates)
  println("k = $k, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = $(power_errors[k]/
 ↪lbda2_over_lbda1^k)")
end
```

```
k = 1, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 116.96410743217001
k = 2, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.737030137537539
k = 3, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.438587254486849
k = 4, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.445295190191076
k = 5, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.368653642725915
k = 6, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.589422174424471
k = 7, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 2.975723858803671
k = 8, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 3.405962453299738
k = 9, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 3.8158987458144766
```

```
k = 10, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 4.17867110482681
… (527 lines omitted) …
k = 538, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08392537613512165
k = 539, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08385015810247844
k = 540, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08381265078431048
k = 541, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08393898875401909
k = 542, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08393504816781458
k = 543, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08383160988847022
k = 544, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08388247115272848
k = 545, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.083796838681471
k = 546, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08389243493660128
k = 547, |lbda^(k)-lbda1| / |lbda2/lbda1|^k = 0.08400203184470283
```

The convergence rate is approximately 0.08.

3. *Implement and test the inverse iteration. Evaluate the convergence rate.*

Notice how the method converges to neighbouring eigenvalues depending on the value of the initial guess mu

[10]:
```
function inverse_iteration(A::AbstractMatrix, q0::Vector{Float64},  ::Float64,␣
↪max_iter::Int=200, tol::Float64=1e-6)
  n = length(q0)
  q = normalize(q0)
  q_hist = Matrix{Float64}(undef, n, max_iter + 1)
  lbda_hist = zeros(Float64, max_iter)

  q_hist[:, 1] = q
  lbda_old = 0.0
  A_shifted = A -   * I
  k = 0
  while k < max_iter
    k += 1
    z = A_shifted \ q
    q = normalize(z)
    q_hist[:, k + 1] = q
    lbda = dot(q, A * q)
    lbda_hist[k] = lbda
    if abs(lbda - lbda_old) < tol
      break
    end
    lbda_old = lbda
  end

  return lbda_hist[1:k], q_hist[:, 1:(k + 1)]
end;
```

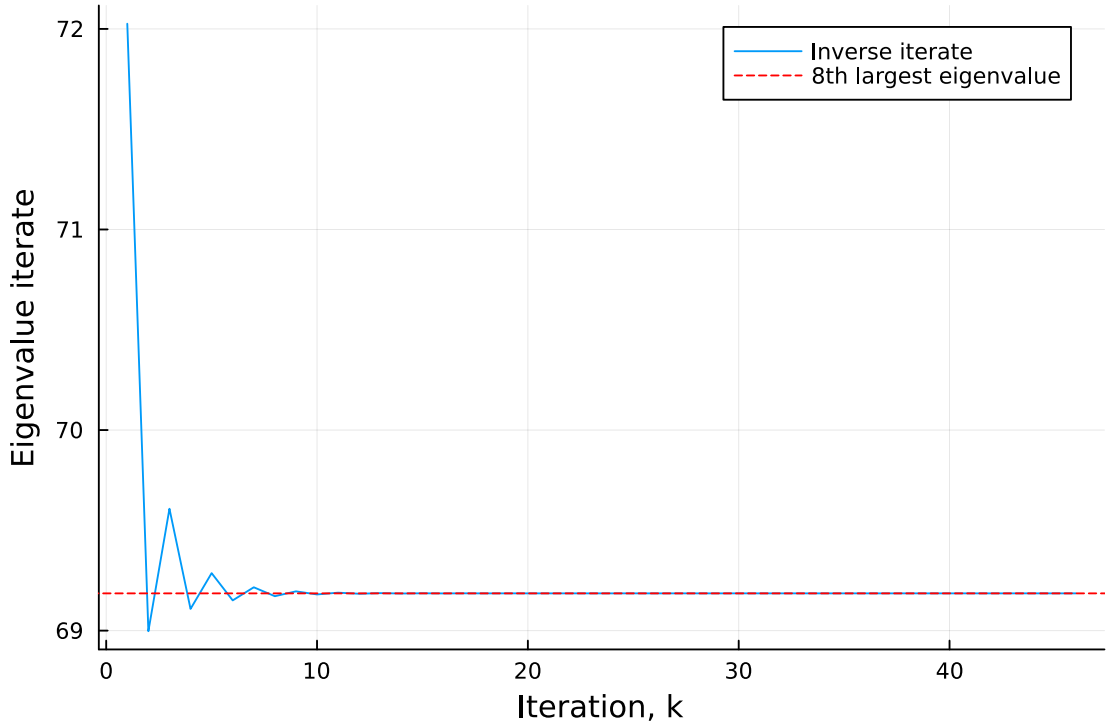
The inverse iteration uses shifted inverse applications to force the iteration towards an arbitrary shift . We can thus use inverse iteration to evaluate any eigenvalue of our choosing.

Say we target the 8th largest eigenvalue lbda8 = 69.185728753422 with an approximation mu = 72.

[11]:
```
mu = 72.;
inverse_iterates, _ = inverse_iteration(A, q0, mu, 1_000, 1e-10);
```

[12]:
```
lbda8 = prescribed_eigvals[end-7]
plot(inverse_iterates, label="Inverse iterate", xlabel="Iteration, k",␣
  ↪ylabel="Eigenvalue iterate")
hline!([lbda8], label="8th largest eigenvalue", linestyle=:dash, color=:red)
```

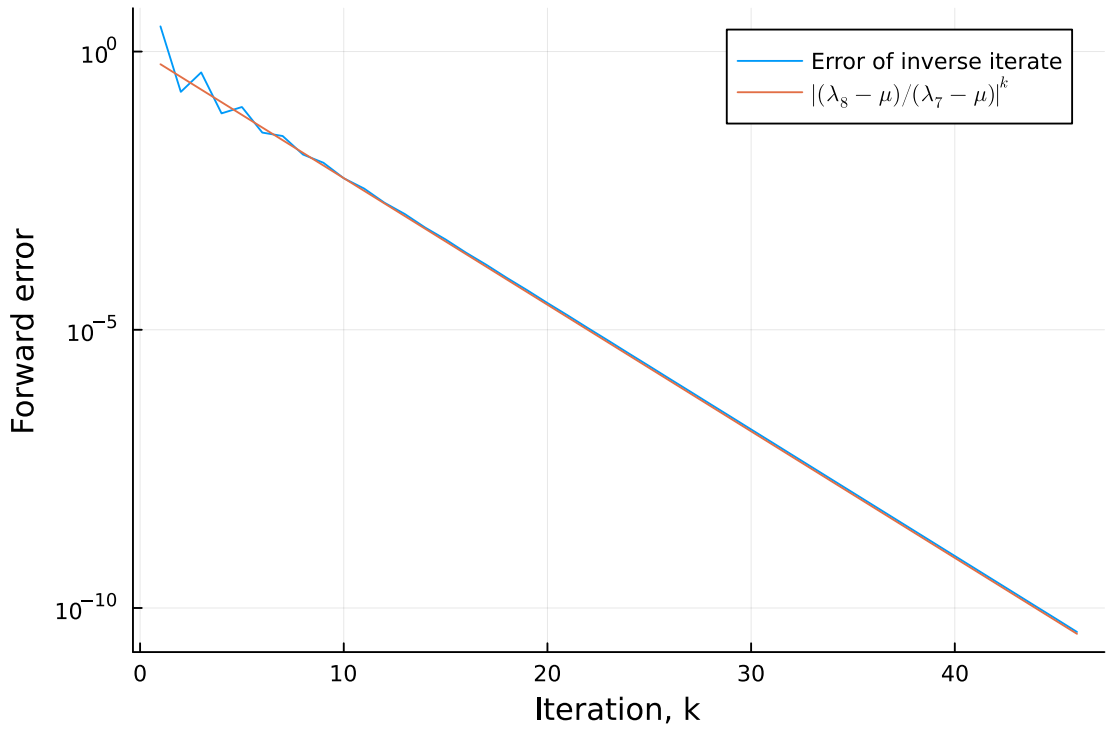
[12]:



We know that the forward error of the inverse iterate converges in $O(r^k)$ where $r := |(\lambda_i - \mu)/(\lambda_j - \mu)|$ in which λ_i and λ_j are the closest and second closest eigenvalues to the shift μ.

[13]:
```
lbda7 = prescribed_eigvals[end-6]
r = abs((lbda8 - mu) / (lbda7 - mu))
inverse_errors = abs.(inverse_iterates .- lbda8)
plot(inverse_errors, yscale=:log10, xlabel="Iteration, k", ylabel="Forward␣
  ↪error", label="Error of inverse iterate")
plot!([r^k for k in 1:length(inverse_iterates)], label=L"|(\lambda_8-\mu)/
  ↪(\lambda_7-\mu)|^k")
```

[13]:



[14]:
```
for k in 1:length(inverse_iterates)
  println("k = $k, |lbda^(k)-lbda8| / r^k = $(inverse_errors[k]/r^k)")
end
```

```
k = 1, |lbda^(k)-lbda8| / r^k = 4.795696556722237
k = 2, |lbda^(k)-lbda8| / r^k = 0.5378751328138058
k = 3, |lbda^(k)-lbda8| / r^k = 2.031070906984177
k = 4, |lbda^(k)-lbda8| / r^k = 0.6295167607984304
k = 5, |lbda^(k)-lbda8| / r^k = 1.3840780425712345
k = 6, |lbda^(k)-lbda8| / r^k = 0.8108711921901053
k = 7, |lbda^(k)-lbda8| / r^k = 1.1870188841964346
k = 8, |lbda^(k)-lbda8| / r^k = 0.9311490542568153
k = 9, |lbda^(k)-lbda8| / r^k = 1.1200889083499135
k = 10, |lbda^(k)-lbda8| / r^k = 0.9953446644570524
… (26 lines omitted) …
k = 37, |lbda^(k)-lbda8| / r^k = 1.0827140293357538
k = 38, |lbda^(k)-lbda8| / r^k = 1.082451694919823
k = 39, |lbda^(k)-lbda8| / r^k = 1.0827608906196797
k = 40, |lbda^(k)-lbda8| / r^k = 1.082463888098693
k = 41, |lbda^(k)-lbda8| / r^k = 1.0828396212085338
k = 42, |lbda^(k)-lbda8| / r^k = 1.0823957167183857
k = 43, |lbda^(k)-lbda8| / r^k = 1.082990761715386
k = 44, |lbda^(k)-lbda8| / r^k = 1.0818410049480787
k = 45, |lbda^(k)-lbda8| / r^k = 1.0839696982537896
```

```
k = 46, |lbda^(k)-lbda8| / r^k = 1.0795892111213121
```

The convergence rate is approximately 1.08.

4. *Implement and test Rayleigh quotient iteration.*

[15]:
```
function RQ_iteration(A::AbstractMatrix, q0::Vector, mu::Number, max_iter::
 ↪Int=200, tol::Float64=1e-6)
  n = length(q0)
  q = normalize(q0)
  q_hist = Matrix(undef, n, max_iter + 1)
  lbda_hist = Vector(undef, max_iter + 1)

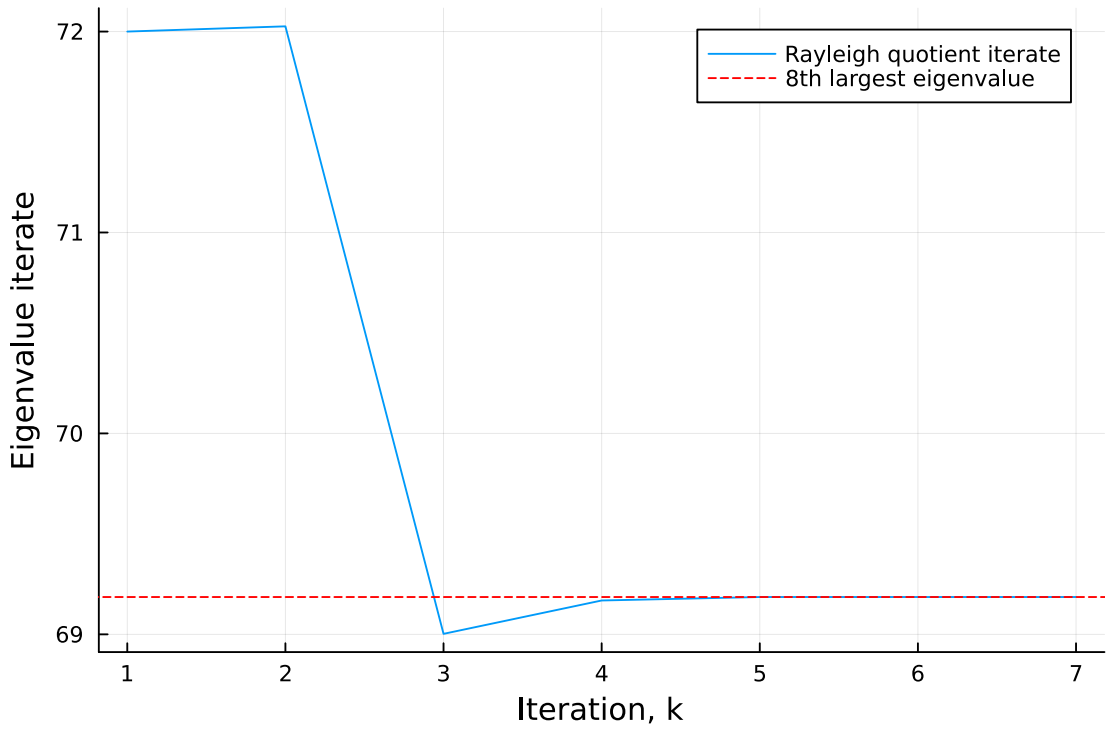
  lbda = mu
  q_hist[:, 1] = q
  lbda_hist[1] = lbda
  k = 0
  while k < max_iter
    k += 1
    lbda_old = lbda
    z = (A - lbda * I) \ q
    q = normalize(z)
    lbda = dot(q, A * q)
    q_hist[:, k + 1] = q
    lbda_hist[k + 1] = lbda
    if abs(lbda - lbda_old) < tol
      break
    end
  end

  return lbda_hist[1:k+1], q_hist[:, 1:k+1]
end;
```

[16]:
```
RQ_iterates, _ = RQ_iteration(A, q0, mu, 1_000, 1e-10);
```

[17]:
```
plot(RQ_iterates, label="Rayleigh quotient iterate", xlabel="Iteration, k",␣
 ↪ylabel="Eigenvalue iterate")
hline!([lbda8], label="8th largest eigenvalue", linestyle=:dash, color=:red)
```

[17]:



We see that the Rayleigh quotient iteration converges even faster than the inverse iterate.