Algorytmy w inżynierii danych

Metody wyznaczania wartości własnych

Wykonał:

Jarosław Królik
[284363] (grupa poniedziałek)

Mateusz Derszniak
[284293] (grupa piątek)

Warszawa 2020

1. Algorytmy

- Używane biblioteki w algorytmach

```
using LinearAlgebra
using Plots

```

- Wywołanie algorytmu metody potęgowej

```
function start_power(A, print_it=false)
    d=typeof(A)
    if (d!=Matrix{Float64})
        A=convert(Matrix{Float64},A)
    end
   return power_metod(A, print_it)
end
```

- Ciało algorytmu metody potęgowej

```
function start_power(A, print_it=false)
    d=typeof(A)
    if (d!=Matrix{Float64})
        A=convert(Matrix{Float64},A)
    end
   return power_metod(A, print_it)
end

function power_metod(A::Matrix{T}, print_it=false, iteration=nothing, delta=10
    dimeansions::Int32 = size(A)[1]
    i::Int32 = size(A)[2]
    if (i < dimeansions)
        dimeansions=i
    end
    A=A[1:dimeansions,1:dimeansions]

    eigen_vector = ones(dimeansions,1)
    eigen_value = norm(eigen_vector)
    if iteration==nothing iteration=10000 end
    last_value::T = 0
     @inbounds for k in 1:iteration
        eigen_vector = A*eigen_vector./eigen_value
        eigen_value = norm(eigen_vector)
        # chech if enought
        if abs(last_value-eigen_value)<delta print_it&&println(k); break end
        last_value=eigen_value
    end
    return Eigen([eigen_value], eigen_vector)
end
```

- Wywołanie algorytmu QR

```
function start_qr(A::Matrix, print_it=false)
    if (typeof(A)!=Matrix{Float64})
        A=convert(Matrix{Float64},A)
    end
   #return E_qr=qr_q_eigen(A)
   return qr_q_eigen(A, print_it)
end
```

- Ciało algorytmu QR

```
function qr_q_eigen(A::Matrix{T}, print_iteration=false, delta=10^-8) where {T<:Float64}
    dim::Int32 = size(A)[1]
    i::Int32 = size(A)[2]
    if (i < dim)
        dim=i
    end
    A=A[1:dim,1:dim]
    A_cpy = A
    Q = Matrix{T}(undef,dim,dim)
    last_value::T = last(A)
    for k in 1:10000
        # A-(QR)->Q
        @inbounds for col in 1:dim
           Q[:,col] =(A[:,col])
           @inbounds for col_proj in 1:(col-1)
               Q[:,col] -= (Q[:,col_proj]'*A[:,col]) / (Q[:,col_proj]'*Q[:,col_proj]) * Q[:,col_proj]
           end
        end
        @inbounds for col in 1:dim
           Q[:,col] /= sqrt(sum(Q[:,col].^2))
        end
        # calc eigen
        A = Q'*A*Q
        # chech if enought
        if abs(last(A)-last_value)<delta print_iteration&&println(k); break end
        last_value=last(A)

    end
    eigenvalue = Vector{T}(undef, dim)
    eigenvector = Matrix{Float64}(undef,dim,dim)
    @inbounds for k in 1:dim
        eigenvalue[k] = A[k,k]
        eigenvector[:,k] = (A_cpy - Array{Float64,2}(I,dim,dim)*A[k,k])*ones(dim,1)
        eigenvector[:,k] /= max(broadcast(abs, eigenvector[:,k])...)
    end
    return Eigen([eigenvalue], eigenvector)
end
```

- Wywołanie algorytmu Householder tranformation

```
function start_house(A::Matrix, print_it=false)
    d=typeof(A)
    if (d!=Matrix{Float64})
        A=convert(Matrix{Float64},A)
    end
   return qr_house_eigen!(A, print_it)
end
```

- Ciało algorytmu Householder tranformation

```
function qr_house_eigen!(A::Matrix{T}, print_iteration::Bool=false, delta=10^-8) where {T<:Float64}
    m:: Int32,n:: Int32 = size(A)
    if (m>n)
     m=n
    end
    if (n>m)
        n=m
    end
    A=A[1:m,1:n]
    #Q = Typeof(A)(I, m,n)
    Q = Array{T,2}(I,m,n)
    for k in 1:n
        z = A[k:m, k]
        v = [ -sign(z[1])*norm(z) - z[1]; -z[2:end] ]
        v /= sqrt(v'*v)
        #for j in 1:n
        #    A[k:m, j] = A[k:m, j] - v*( 2*(v'*A[k:m,j]) )
        #end
        for j in 1:m
            Q[k:m, j] = Q[k:m, j] - v*( 2*(v'*Q[k:m,j]) )
        end
    end
    Q = Q'
    #R = triu(A)
```

```
    A_cpy = A
    last_value::T = last(A)
    for k in 1:100
        #QR = qr(A)
        #A = QR.Q'*A*QR.Q
        #Q = qr_q(A)
        Q = Array{Float64,2}(I,m,n)
        for k in 1:n
            z = A[k:m, k]
            v = [ -sign(z[1])*norm(z) - z[1]; -z[2:end] ]
            v /= sqrt(v'*v)

            #for j in 1:n
            #    A[k:m, j] = A[k:m, j] - v*( 2*(v'*A[k:m,j]) )
            #end
            for j in 1:m
                Q[k:m, j] = Q[k:m, j] - v*( 2*(v'*Q[k:m,j]) )
            end
        end

        #Q = Q'
        A = Q*A*Q'

        if abs(last(A)-last_value)<delta print_iteration&&println(k); break end
        last_value=last(A)
        #A = R*Q
        #?A = Q*A*Q'
    end
    eigenvalue = Vector{Float64}(undef, m)
    eigenvector = A_cpy
    @inbounds for k in 1:m
        eigenvalue[k] = A[k,k]
        eigenvector[:,k] = (A_cpy - Array{Float64,2}(I,m,n)*A[k,k])*ones(m,1)
        eigenvector[:,k] /= max(broadcast(abs, eigenvector[:,k])...)
    end
    return Eigen([eigenvalue], eigenvector)
    end
```

2. Wynik działania algorytmów

- Algorytm potęgowy

```
foo = [12 -51 4; 6 167 -68; -4 24 -41] * 1.0
```

```
3×3 Array{Float64,2}:
  12.0  -51.0    4.0
   6.0  167.0  -68.0
  -4.0   24.0  -41.0
```

```
E = start_power(foo)
```

```
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
1-element Array{Float64,1}:
 156.1366840619636
vectors:
3×1 Array{Float64,2}:
 -51.23585057736389
 146.28154890724886
  18.848346738524288
```

- Algorytm QR

```
foo = [12 -51 4; 6 167 -68; -4 24 -41] * 1.0
```

```
3×3 Array{Float64,2}:
  12.0  -51.0    4.0
   6.0  167.0  -68.0
  -4.0   24.0  -41.0
```

```
E = start_qr(foo)
```

```
Eigen{Float64,Array{Float64,1},Array{Float64,2},Array{Array{Float64,1},1}}
values:
1-element Array{Array{Float64,1},1}:
 [156.13668406196905, -34.19667500330554, 16.059990941336864]
vectors:
3×3 Array{Float64,2}:
 -1.0       -0.00577115  -0.574095
 -0.26754    1.0          1.0
 -0.926754   0.094806    -0.416685
```

- Algorytm Householder tranformation

```
foo = [12 -51 4; 6 167 -68; -4 24 -41] * 1.0
```

```
3×3 Array{Float64,2}:
  12.0  -51.0    4.0
   6.0  167.0  -68.0
  -4.0   24.0  -41.0
```

```
E = start_house(foo)
```

```
Eigen{Float64,Array{Float64,1},Array{Float64,2},Array{Array{Float64,1},1}}
values:
1-element Array{Array{Float64,1},1}:
 [156.13668406196905, -34.19667500323768, 16.059990941268875]
vectors:
3×3 Array{Float64,2}:
 -1.0       -0.10384   -0.157977
 -0.26754    1.0       -1.0
 -0.926754   0.122395  -0.694421
```

3. Porównanie algorytmów

- Szybkość działania

a) Przygotowanie do testów

```
using BenchmarkTools

mutable struct TestedFunction
    fun::Function
    name::String
    result
    TestedFunction(fun, name) = new(fun, name, nothing)
end

tested_functions = []
push!(tested_functions, TestedFunction(eigen,"default"))
push!(tested_functions, TestedFunction(start_power,"power"))
push!(tested_functions, TestedFunction(start_qr,"qr"))
push!(tested_functions, TestedFunction(start_house,"house"))

for i in 1:size(tested_functions)[1]
    pop!(tested_functions)
end

foo = [12 -51 4; 6 167 -68; -4 24 -41] * 1.0

```

b) Funkcja testująca

```
function bench(a=foo)
    global fun
    for tested_fun in tested_functions
        fun = tested_fun.fun
        b = @benchmarkable (fun($a)) seconds=1.0
        tune!(b)
        tested_fun.result = (run(b))
    end
    print_bench()
end

function print_bench()
    for tested_fun in tested_functions
        println(tested_fun.result)
    end
end
```

c) Wyniki testów

```
julia> bench()
Trial(4.300 μs)
Trial(4.900 μs)
Trial(56.400 μs)
Trial(268.001 μs)
```

```
julia> tested_functions
4-element Array{Any,1}:
 TestedFunction(LinearAlgebra.eigen, "default", Trial(4.300 μs))
 TestedFunction(start_power, "power", Trial(4.900 μs))
 TestedFunction(start_qr, "qr", Trial(56.400 μs))
 TestedFunction(start_house, "house", Trial(268.001 μs))
```

- Obciążenie procesora

  (będą wykonane w następnej części)

- Zajętość pamięciowa

  (będą wykonane w następnej części)