Practica 2

Prestaciones numericas del metodo de la potencia.

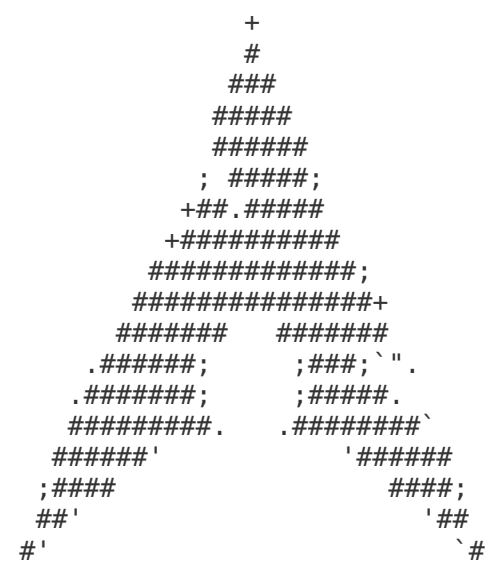
Miguel Garcia Lafuente

Índice de contenido

Para esta practica se ha usado un ordenador portatil con las siguentes caracteristicas:

```
S.O: ArchLinux 64 bits.
Memoria RAM: 8076456 kB. (unos 7.7GB)
CPU: Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz – 64 bits

~ » archey3
```



```
OS: Arch Linux x86_64

Kernel Release: 3.17.4-1-ARCH
Uptime: 9:53
WM: None
DE: None
Packages: 1759
RAM: 3922 MB / 7887 MB
Processor Type: Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz
$EDITOR: vim
Root: 322G / 458G (70%) (ext4)
```

Version de Julia 0.3.3:

```
~ » julia
rock@neurotiko

               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.3.3 (2014-11-23 20:19 UTC)
 _/ |\__'_|_|_|\__'_|  |
|__/                   |  x86_64-unknown-linux-gnu

julia>
```

Como apunte, a pesar de tener 4 cores, Julia esta en estado de crecimiento y por ahora solo funciona en un core, por lo que a efectos practico el procesador es de un solo nucleo a 2.5 GHz.

Toda la practica esta vez lo he pasado tambien con un programa que se llama “Ijulia notebook”, que es un modulo para el “Ipython notebook”, para poder subir codigo, y ejecutar codigo, y que quede en un formato perfecto para presentar.

Como se puede observar, esta todo el codigo, pero tambien las ejecuciones y los resultados.

Link al notebook:

http://nbviewer.ipython.org/github/rockneurotiko/rockneurotiko.github.io/blob/master/Universidad/AplicacionesNumericas/PageRank/Dia2/AplNum-Practica2.ipynb

A. Rutinas Auxiliares

1. Metodo de la potencia

En esta practica usaremos la misma funcion de la potencia que la que se hizo para la practica anterior, pero con los tipos declarados para permitir “Matrices” y “Sparse”, asi como hacer que todos los parametros tengan valor por defecto de forma individual (menos la matriz, claro).

```
function potencia(M::Union(Array{Float64,2},
                           SparseMatrixCSC{Float64,Int64}),
                  tolerancia::Float64=1e-12,
                  n_iter::Int64=500,
                  r::Array{Float64,2}=ones(size(M,1),1))
    i = 0
    n = size(M)[1]
    last = zeros(n) ## Just to initialize
    while i < n_iter && maximum(abs((r/norm(r)) - last)) > tolerancia
        i += 1
        last = r

        last = last ./ norm(last)
        r = M * last
    end
    return norm(r), last, i        # This to return the eigvalues
    # return norm(r),pagerank(last),i # This to return the pagerank
end
```

2. Matrices Completas y Densas

La funcion “mcden” es tan sencillo como hacer esto:

```
1 function mcden(N::Int64)
2     A = rand(N,N)
3     return A./sum(A,1)
4 end
```

Comprobemos que cumple con el teorema de Perron-Frobenius.

Al no tener ningun numero igual a cero, no es reducible.

Comprobemos que es estocastica:

```
julia> sum(l,1)
1x10 Array{Float64,2}:
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Si, lo es, para todas sus columnas, los elementos suman 1.

Y que no es negativa (busca que algun elemento sea menor que cero, y si hay al menos uno devuelve true, si no hay ninguno, devuelve false):

```
julia> any(map(x -> x<0, l))
false
```

3. Matrices Completas y Dispersas.

Julia en la version 0.3.3 no tiene “randi”, en las anteriores tenian, pero decidieron unificar todo en “rand”, pero es muy sencilla de codificar:

```
1 function randi(a,b,c)
2     vec(rand(1:a,b,c))  # Se transforma a vector porque rand devuelve una matriz
3 end
```

Asi, la funcion principal “mcdis” seria asi:

```
 1 function mcdis(N::Int64, R::Int64=5)
 2     p = randi(N,1,5*N)
 3     q = randi(N,1,5*N)
 4     Cs = sparse(p,q,1.0,N,N)
 5     C = full(Cs)
 6     Nj = sum(C,1)
 7     dj = [x == 0 ? 1:0 for x=Nj]
 8     if ! (1 in dj)
 9         return mcdis(N,R)
10     else
11         return C
12     end
13 end
```

Calculemos el porcentaje de no ceros.

```
julia> a = mcdis(500);

julia> no_zero = 0;

julia> for i in a
           if i != 0
               no_zero += 1
           end
       end

julia> no_zero
2483

julia> (no_zero / (500*500)) * 500
4.966
```

Solo un 4.966 % de numeros distintos de ceros en 250.000 elementos.

Calculemos el tamaño ocupado en memoria RAM:

```
julia> sizeof(a)
2000000
```

1953.125 KB (1.90 MB) por una matriz 500x500.

Creemos la matris S.

```
julia> b = create_S(a);
```

Comprobamos que es estocastica.

```
julia> any(map(x->x==0,sum(b,1)))
false
```

Y que no es negativa.

```
julia> any(map(x -> x<0, b))
false
```

4. Matrices Dispersas.

El codigo para generar matrices dispersas es igual que la funcion para crear matrices dispersas completas, solo que sin transformarlo a matriz total, por lo que la matriz es Sparse.

```
 1 function mdis(N::Int64, R::Int64=5)
 2     p = randi(N,1,5*N)
 3     q = randi(N,1,5*N)
 4     C = sparse(p,q,1.0,N,N)
 5     Nj = sum(C,1)
 6     dj = [x == 0 ? 1:0 for x=Nj]
 7     if ! (1 in dj)
 8         return mdis(N,R)
 9     else
10         return C
11     end
12 end
```

Para calcular la memoria ocupada en “SparseMatrix” hay que hacer un truquillo, ya que si pedimos el tamaño nos da el basico de una matriz Sparse:

```
julia> a = mdis(500);

julia> sizeof(a)
40
```

Pero, con los tipos de la matriz:

```
julia> typeof(a)
SparseMatrixCSC{Float64,Int64}
```

Sabemos que la matriz Sparse que estamos creando contiene dos enteros de 64 bits y un float de 64 bits por cada entrada, asi que la funcion para saber el tamaño ocupado es la siguiente:

```
1 function sizesp(a::SparseMatrixCSC{Float64,Int64})
2     sizeof(a) +  # Tamaño base de SparseMatrix
3      (sizeof(Float64) * size(a,1)) + # Tamaño de los "n" elementos
4     (sizeof(Int64) * size(a,1) * 2)  # Tamaño de los dos indices
5 end
```

Asi que para nuestra matriz de 500x500 que hemos creado:

```
julia> sizesp(a)
12040
```

Es decir, 11.75 KB.

Para crear la matriz S se podria usar la misma funcion que hemos usado antes (que esta descrita en la practica anterior), pero despues de muchas pruebas y mucho codigo intentado, este es el mejor codiga para transformar una SparseMatrix en su correspondiente S, en Sparse todavia.

Primero, “normalizamos” la matriz, es decir, que si alguna columna suma cero, vamos a añadir 1/n en todos los valores, lo mas importante de esto es que crea una nueva matriz “sparse”.

```
function normalize_sp(Cs,n=size(Cs,1))
    j,i = findn(Cs)
    dj = [x == 0 ? 1:0 for x=sum(Cs,1)]
    I = findn(dj)[1]
    for idx in I
        append!(i,[idx for _ in range(1,n)])
        append!(j,range(1,n))
    end
    sparse(j,i,1.0,n,n)
end
```

Una vez tenemos una matriz sparse “normalizada”, podemos aplicarle el siguiente metodo para crear S, en “sparse”:

```
function create_S_normalized(A :: SparseMatrixCSC,n)
    sums = sum(A,1)
    i,j = findn(A)
    I,J,V = findnz(A)
    for idx in 1:length(V)
        V[idx] /= sums[J[idx]]
    end
    sparse(i,j,V,n,n)
end
```

Asi que, creemos S:

```
julia> a = mdis(500);

julia> b = normalize_sp(a,500);

julia> b = create_S_normalized(b,500);
```

Comprobemos que es estocastica y no negativa:

```
julia> any(map(x->x==0,sum(b,1)))
false

julia> any(map(x -> x<0, b))
false
```

B. Prestaciones numericas del metodo de la potencia.

Voy a usar dos funciones auxiliares para calcular las dos precisiones:

```
function prec1(A,x)
    norm(abs(A*x-x))
end

function prec2(lambda)
    abs(lambda-1.0)
end
```

Debido a que la matriz S se calcula distinto (para las matrices “Sparse” hay que “normalizar” primero) se van a usar dos funciones de test distintas.

Esta primera funcion es para la matrices completas densas y las dispersas:

La principal diferencia notable con lo propuesto es que se hace uso de la macro @time, en lugar de usar “tic” y “toc” (que tambien lo tiene), ya que la macro @time nos va a dar el tiempo exacto de aplicación de una funcion, asi como la memoria que ha necesitado “allocar”. Tambien, lo aplico a las 3 partes criticas, la creacion de la matriz, la transformacion a S y por ultimo el metodo de la potencia en si.

Tambien, notese que la funcion que crea la funcion se pasa por parametro, ya que Julia tiene funciones de primer orden, se pueden pasar como parametro.

```
function test(N::Int64,
              create::Function=mcden)
    S = @time create(N)
    @time create_S!(S,N)
    tol=1e-13
    @time lambda, x, iter = potencia(S,tol)
    p1 = prec1(S,x)
    p2 = prec2(lambda)
    return p1, p2, sizeof(S), iter
end
```

El codigo para las matrices “Sparse” es el siguiente:

```
function test_sp(N::Int64)
    S = @time mdis(N)
    @time S = normalize_sp(S,N)
    @time S = create_S_normalized(S,N)
    tol=1e-13
    @time lambda, x, iter = potencia(S,tol)
    p1 = prec1(S,x)
    p2 = prec2(lambda)
    return p1, p2, sizesp(S), iter
end
```

Como vemos es muy parecido, la unica diferencia es que normalizamos antes (y le hacemos el “time” tambien), y el tamaño se calcula con la funcion creada antes.

- Completar las tablas para Matrices Completas y Densas.

El tiempo se va a considerar solo el que tarda en aplicar la potencia, ya que es el que esta entre el “tic-toc” del codigo proporcionado.

```
julia> test(10^1)
elapsed time: 1.3224e-5 seconds (2568 bytes allocated)
elapsed time: 4.87e-6 seconds (3216 bytes allocated)
elapsed time: 1.8817e-5 seconds (13440 bytes allocated)
(1.1746701648122892e-13,8.215650382226158e-15,800,18)

julia> test(10^2)
elapsed time: 0.000168153 seconds (161640 bytes allocated)
elapsed time: 8.7223e-5 seconds (182784 bytes allocated)
elapsed time: 0.000213637 seconds (54592 bytes allocated)
(2.654886459713708e-14,2.220446049250313e-16,80000,11)

julia> test(10^3)
elapsed time: 0.014521225 seconds (16009128 bytes allocated)
elapsed time: 0.005369095 seconds (16176464 bytes allocated)
elapsed time: 0.006263743 seconds (323392 bytes allocated)
(6.915008764401647e-13,1.1102230246251565e-16,8000000,7)

julia> test(10^4)
elapsed time: 1.235342936 seconds (1600081112 bytes allocated, 2.46% gc time)
elapsed time: 0.947138254 seconds (1601440448 bytes allocated, 48.01% gc time)
elapsed time: 0.464823106 seconds (2802416 bytes allocated)
(3.759542748591992e-14,2.4424906541753444e-15,800000000,6)

julia> test(10^5)
ERROR: MemoryError()
 in mcden at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:100
 in test at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:56
 in test at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:151
```

	Tiempo	Memoria	Iteraciones	Precision1	Precision2
N=10^1	1.8817e-5 s	800 B	18	1.174670e-13	8.215650e-15
N=10^2	0.000213637 s	78,125 KB	11	2.654886e-14	2.220446e-16
N=10^3	0.006263743 s	7.62939 MB	7	6.91500e-13	1.1102230e-16
N=10^4	0.464823106 s	762,93 MB	6	3.759542e-14	2.4424906e-15

Gracias a la siguiente funcion vamos a calcular mas o menos cuanto vale la constante:

```
1 function calc_alpha(t, N)
2     t / (N ^ 2)
3 end
```

```
julia> calc_alpha( 1.8333e-5, 10^1)
1.8333e-7

julia> calc_alpha( 0.000213637, 10^2)
2.13637e-8

julia> calc_alpha( 0.006263743, 10^3)
6.263743e-9

julia> calc_alpha( 0.464823106, 10^4)
4.6482310599999995e-9
```

	Tiempo Estimado	Memoria Estimada
N=10^5	1 minuto.	74.50 GB
N=10^6	1.7 horas.	7.27 TB
N=10^10	2.794 siglos.	693.889 EB (ExaBytes)

- Completar las tablas para Matrices Completas y Dispersas.

Se usa la misma funcion test, solo que se le pasa como argumento la funcion que crea matrices dispersas.

```
julia> test(10^1, mcdis)
elapsed time: 0.000135142 seconds (100736 bytes allocated)
elapsed time: 4.167e-6 seconds (2936 bytes allocated)
elapsed time: 2.2316e-5 seconds (19072 bytes allocated)
(7.759764525326414e-14,3.885780586188048e-15,800,26)

julia> test(10^2, mcdis)
elapsed time: 0.000168176 seconds (231408 bytes allocated)
elapsed time: 0.000106452 seconds (177336 bytes allocated)
elapsed time: 0.0006039 seconds (204544 bytes allocated)
(2.692753305861874e-13,8.881784197001252e-15,80000,44)

julia> test(10^3, mcdis)
elapsed time: 0.002541218 seconds (8338272 bytes allocated)
elapsed time: 0.026035049 seconds (16047120 bytes allocated, 76.14% gc time)
elapsed time: 0.035948963 seconds (1737952 bytes allocated)
(3.853389157729466e-13,1.354472090042691e-14,8000000,42)

julia> test(10^4, mcdis)
elapsed time: 0.165374512 seconds (803362304 bytes allocated, 9.32% gc time)
elapsed time: 0.944097135 seconds (1590711336 bytes allocated, 54.36% gc time)
elapsed time: 3.242376086 seconds (17214512 bytes allocated)
(4.159343751248869e-13,6.661338147750939e-16,800000000,42)

julia> test(10^5, mcdis)
ERROR: MemoryError()
 in full at sparse/sparsematrix.jl:173
 in mcdis at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:108
 in mcdis at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:105
```

	Tiempo	Memoria	Iteraciones	Precision1	Precision2
N=10^1	2.8163e-5 s	800 B	26	7.759764e-14	3.885780e-15
N=10^2	0.0006039 s	78,125 KB	44	2.692753e-13	8.881784e-15
N=10^3	0.035948963 s	7.62939 MB	42	3.853389e-13	1.354472e-14
N=10^4	3.242376086 s	762,93 MB	42	4.159343e-13	6.661338e-16

```
julia> calc_alpha( 2.8163e-5, 10^1)
2.8163e-7

julia> calc_alpha(0.0006039, 10^2)
6.039e-8

julia> calc_alpha(0.035948963, 10^3)
3.5948963e-8

julia> calc_alpha(3.242376086 , 10^4)
3.242376086e-8
```

	Tiempo Estimado	Memoria Estimada
N=10^5	5,4 minutos.	74.50 GB
N=10^6	9.4 horas.	7.27 TB
N=10^10	79.84 siglos.	693.889 EB (ExaBytes)

- Completar las tablas para Matrices Dispersas.

```
julia> test_sp(10^1)
elapsed time: 6.892e-5 seconds (25936 bytes allocated)
elapsed time: 1.2411e-5 seconds (7192 bytes allocated)
elapsed time: 2.0692e-5 seconds (9144 bytes allocated)
elapsed time: 3.7418e-5 seconds (17552 bytes allocated)
(8.893696902501042e-14,3.774758283725532e-15,280,25)

julia> test_sp(10^2)
elapsed time: 9.7915e-5 seconds (55808 bytes allocated)
elapsed time: 7.006e-5 seconds (74728 bytes allocated)
elapsed time: 0.000161018 seconds (99192 bytes allocated)
elapsed time: 0.000234202 seconds (252752 bytes allocated)
(1.20002830208126e-13,1.3766765505351941e-14,2440,55)

julia> test_sp(10^3)
elapsed time: 0.000969191 seconds (561168 bytes allocated)
elapsed time: 0.000836056 seconds (1325656 bytes allocated)
elapsed time: 0.002770855 seconds (2013144 bytes allocated)
elapsed time: 0.001995793 seconds (1735920 bytes allocated)
(2.641212931895892e-13,9.103828801926284e-15,24040,42)

julia> test_sp(10^4)
elapsed time: 0.009282047 seconds (5744976 bytes allocated)
elapsed time: 0.045665501 seconds (50859080 bytes allocated, 47.46% gc time)
elapsed time: 0.119411382 seconds (91060776 bytes allocated, 32.59% gc time)
elapsed time: 0.039307626 seconds (16812192 bytes allocated)
(6.223063329950647e-13,3.774758283725532e-15,240040,41)

julia> test_sp(10^5)
elapsed time: 0.094577652 seconds (57584960 bytes allocated, 16.44% gc time)
elapsed time: 3.87985206 seconds (8962042536 bytes allocated, 4.95% gc time)
elapsed time: 21.139255319 seconds (13070329112 bytes allocated, 15.68% gc time)
elapsed time: 4.833984076 seconds (188013632 bytes allocated, 0.50% gc time)
(1.1874959501967848e-13,8.881784197001252e-16,2400040,46)

julia> test_sp(2*10^5)
elapsed time: 0.240790054 seconds (115187840 bytes allocated, 7.25% gc time)
ERROR: MemoryError()
 in sparse at sparse/csparse.jl:38
 in normalize_sp at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/sparse_potencia.jl:100
 in test_sp at
/home/rock/Git/rockneurotiko.github.io/Universidad/AplicacionesNumericas/PageRank/
Dia2/potencia.jl:56
```

	Tiempo	Memoria	Iteraciones	Precision1	Precision2
N=10^1	3.7418e-5 s	280 B	25	8.893696e-14	3.77475e-15
N=10^2	0.000234202 s	2.38 MB	55	1.20002e-13	1.3766765e-14
N=10^3	0.001995793 s	23.47 MB	42	2.6412129e-13	9.1038288e-15
N=10^4	0.039307626 s	234.41 MB	41	6.2230633e-13	3.7747582e-15
N=10^5	4.833984076 s	2.28 GB	46	1.1874959e-13	8.881784e-16

```
julia> calc_alpha(3.7418e-5, 10^1)
3.7418e-7

julia> calc_alpha(0.000234202, 10^2)
2.34202e-8

julia> calc_alpha(0.001995793, 10^3)
1.995793e-9

julia> calc_alpha(0.039307626, 10^4)
3.9307626e-10

julia> calc_alpha(4.833984076, 10^5)
4.833984076e-10
```

	Tiempo Estimado	Memoria Estimada
N=10^6	8.05 minutos	22.88 GB
N=10^7	13 horas.	228.88 GB
N=10^10	1.19 siglos.	223.51 TB

C. Conclusiones finales

1. ¿Mayor limitacion o dificultades?

Bajo mi punto de vista, la mayor limitacion es la memoria RAM necesaria para guardar las matrices (completas o dispersas), mucho mas en matrices completas, por supuesto.

El tiempo de procesamiento se podria reducir mucho si la computacion fuese de forma paralela en un cluster.

Por ejemplo, mi single-core (julia usa solo un core, pero se puede paralelizar) de 2.5GHz son, teoricamente, unos 10GFLOPS.

Usemos las especificaciones del Magerit 2.0, son 245 nodos con 16 cores cada nodo, y 32 GB de RAM. Cada core de cada nodo alcanza 18.38 GFLOPS.

El calculo es sencillo:

```
julia> 18.38 * 16 * 245
72049.59999999999
```

Es decir, 72049 GFLOPS, que proporcionalmente a nuestros 10GFLOPS es 7204.95999 veces mas potente.

Por lo que, teoricamente, en matrices dispersas, el 1.19 siglo que tardabamos en calcular para N=10^10, en Magerit 2.0 se tardarian unas 6,03 horas.

Sin embargo, con los 32GB de RAM por cada nodo, tendriamos 7.65 TB de memoria RAM en total,

que no dan para los 223 TB que hacen falta, por lo que tendriamos que idear un sistema que repartiese los datos en todas las maquinas en memoria RAM + Disco duro y que mediante pasos de mensajes accediesen a todos los datos como si fuese uno, que, gracias a los 0.3 microsegundos de latencia seria casi imperceptible a nivel de red.

2. Cuantificar la mejora de las matrices dispersas respecto a las completas.

Cuantifiquemos uno que hemos podido ejecutar, y luego uno teorico.

Para N=10^4:

- Dispersas vs Completas Densas:

Tiempo:

```
julia> 0.464823106 / 0.039307626
11.825265306024841
```

11 veces mas rapido con matrices dispersas.

Memoria:

```
julia> 762.93 / 234.41
3.2546819674928544
```

Las matrices dispersas usan 3 veces menos memoria RAM que las completas densas.

- Dispersas vs Completas Dispersas.

Tiempo:

```
julia> 3.2546819674928544 / 0.039307626
82.80026800633685
```

Las matrices dispersas son 82 veces mas rapido que las completas dispersas. Esto es debido a que las Completa Dispersas necesita bastantes mas iteraciones para llegar a la precision indicada.

Memoria:

```
julia> 762.93 / 234.41
3.2546819674928544
```

Debido a que usan la misma memoria las Completas Dispersas que las Completas Densas, la diferencia es la misma, 3 veces menos.

Teorico, para N=10^10

- Dispersas vs Completas Densas:

Tiempo:

```
julia> 8.811158399999998e9 / 3.7542072068203287e9
2.3470090793051126
```

2 veces mas rapido con matrices dispersas.

Memoria:

```
julia> 7.27595352064e8 / 223.51
3.2553145365487006e6
```

Las matrices dispersas usan 3*10^6 veces menos memoria RAM que las completas densas, es decir, una barbaridad menos.

- Dispersas vs Completas Dispersas.

Tiempo:

```
julia> 2.51783424e11 / 3.7542072068203287e9
67.06700246661426
```

Las matrices dispersas son 67 veces mas rapido que las completas dispersas en este caso. Esto es debido a que las Completa Dispersas necesita bastantes mas iteraciones para llegar a la precision indicada.

Memoria:

```
julia> 7.27595352064e8 / 223.51
3.2553145365487006e6
```

Debido a que usan la misma memoria las Completas Dispersas que las Completas Densas, la diferencia es la misma, 3*10^6 veces menos.

3. ¿Que relacion ajusta mejor los datos?

4. Estimar el coste computacional del metodo de la potencia.

A partir de aquí, n es igual al tamaño del vector “r”.

Operaciones:

En la comparacion de la condicion del while:

- 1 para la comparacion de numero de iteraciones.

- 1 para saber el maximo.

- n para dividir el vector r entre su norma.

- n para calcular su norma.

- n para restar el resultado entre el anterior.

- 1 para compararlo con la tolerancia.

Coste condicion while = 3 + 3n

Dentro del while:

- 1 para aumentar la iteracion.

- 1 para asignar “last” a “r”.

- n para calcular la norma de “last”.

- n para dividir “last” entre la norma calculada antes.

- 1 para asignar la division calculada antes.

- n*n para multiplicar “M” por “last”.

Coste del bucle interno: 3 + 2n + n^2.

Estos dos costes son “m” veces, donde “m” es el numero de iteraciones.

Por ultimo hay “n” mas para calcular la ultima norma.

El coste estimado seria:

$$m(6+5n+n^2)+n$$

m = numero de iteraciones.

n = tamaño del vector “r”.

5. Hacer cualquier observacion con relevancia numerica.

Aparte de las observaciones hechas durante todo el documento, quiero remarcar una ultima cosa.

Al haber puesto un indicador de tiempo en todas las operaciones, tanto en la creacion de la matriz aleatoria, como en la transformacion de esa matriz a la matriz S que cumple Perron-Frobenius, podemos ver, claramente, que, excepto en las matrices completas dispersas, se tarda mas en transformar la matriz a la matriz S que en aplicar el metodo de la potencia.

Lo ideal seria, a la hora de conseguir los datos de “links”, que en lugar de tener solamente el link (con el sentido), que seria, por ejemplo, algo asi:

```
2 3
2 4
3 1
4 5
5 6
6 2
6 3
6 4
```

Tener el nodo, y el numero de links de salida, y luego los links. Con esto, los datos anteriores quedan asi:

```
2 2
3
4
3 1
1
4 1
5
5 1
6
6 3
2
3
4
```

Con esto, lo que consigues, es que a la hora de la creacion de la matriz, en lugar de inicializarla con “1” y luego transformarlo a S, eres capaz de crear la matriz S según leas los datos, lo cual te ahorra muchisimo calculo.

Tambien, aunque no lo se al 100%, seguro que hay algun metodo para no tener que hacer que las columnas que no tiene ningun elemento sean 1/n, haciendo que en cada iteracion, se “acarree” algun valor, ya que todas esas columnas tienen el mismo peso y afectan igual a todos los nodos, y cuando termina (o en cada paso, no lo se exactamente), se le suma el acarreo al nodo, por lo que no hace tener mas memoria, ni hacer iteraciones para asignar ese valor a toda la columna.