MAP 2112 – Introdução à Lógica de Programação e Modelagem Computacional

1º Semestre - 2019

Prof. Dr. Luis Carlos de Castro Santos

lsantos@ime.usp.br/lccs13@yahoo.com

Algoritmo

Método do Ponto Fixo: dados uma aproximação inicial p_0, uma tolerância $TOL > 0$ e o número máximo de iterações N_0, devolve a solução aproximada p ou uma mensagem de erro.

Passo 1: Faça $k \leftarrow 1$.

Passo 2: Enquanto $k \leq N_0$, execute os passos 3 a 6:

Passo 3: Faça $p \leftarrow g(p_0)$.

Passo 4: Se $|p - p_0| < TOL$ ou $\frac{|p-p_0|}{|p|} < TOL$ ou $|f(p)| < TOL$, então devolva p como solução e pare.

Passo 5: Faça $k \leftarrow k + 1$.

Passo 6: Faça $p_0 \leftarrow p$.

Passo 7: Escreva "o método falhou após N_0 iterações" e pare.

```r
#metodo do ponto fixo
f <- function(x){x^3+4*x^2-10}
g <- function(x){sqrt(10/(4+x))}
x0 <- 1.5
tol <- 10^-5
n0 <- 100

pontofixo <- function(f,g,p0,tol,n0){
             eps <- 1
             p <- p0
             k <- 0
             while(eps > tol){
               p <- g(p)
               eps <- abs(f(p))
               if (k >= n0) {
                 print('No. max. iterações atingido')
                 break}
               k <- k + 1

             print('iteração')
             print(k)
             print('erro')
             print(eps)
             print('raiz')
             print(p)
             }
             p
}

pontofixo(f,g,x0,tol,n0)
```

$$x^3 + 4x^2 - 10 = 0$$

$$x = g_4(x) = \sqrt{\frac{10}{4+x}};$$

Percebi agora que minha implementação não segue exatamente o algoritmo apresentado, como exercício modifique o meu algoritmo para ele coincida com a forma proposta.

```
> source('C:/Users/User/Dropbox/USP/2019/MAP2112/Notas de Aula/ponto_fixo.R')
[1] "iteração"
[1] 1
[1] "erro"
[1] 0.2756369
[1] "raiz"
[1] 1.3484
[1] "iteração"
[1] 2
[1] "erro"
[1] 0.03548098
[1] "raiz"
[1] 1.367376
[1] "iteração"
[1] 3
[1] "erro"
[1] 0.004507522
[1] "raiz"
[1] 1.364957
[1] "iteração"
[1] 4
[1] "erro"
[1] 0.0005735977
[1] "raiz"
[1] 1.365265
[1] "iteração"
[1] 5
[1] "erro"
[1] 7.297674e-05
[1] "raiz"
[1] 1.365226
[1] "iteração"
[1] 6
[1] "erro"
[1] 9.284815e-06
[1] "raiz"
[1] 1.365231
>
```

Método do Ponto Fixo - exemplo

Com $p_0 = 1.5$, a tabela a seguir mostra os resultados da aplicação do Método do Ponto Fixo para as 5 opções de g. A raiz verdadeira é 1.365230013.

k	g_1	g_2	g_3	g_4	g_5
0	1.500	1.5000	1.500000000	1.500000000	1.500000000
1	-0.875	0.8165	1.286953768	1.348399725	1.373333333
2	6.732	2.9969	1.402540804	1.367376372	1.365262015
3	-469.700	$\sqrt{-8.65}$	1.345458374	1.364957015	1.365230014
4	1.03×10^8		1.375170253	1.365264748	1.365230013
5			1.360094193	1.365225594	
6			1.367846968	1.365230574	
7			1.363887004	1.365229942	
8			1.365916734	1.365230022	
9			1.364878217	1.365230012	
10			1.365410062	1.365230014	
15			1.365223680	1.365230013	
20			1.365230236		
25			1.365230006		
30			1.365230013		

Introduction to the R Language

Loop Functions

Roger D. Peng, Associate Professor of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Looping on the Command Line

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

lapply

`lapply` takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its ... argument. If `X` is not a list, it will be coerced to a list using `as.list`.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x7ff7a1951c00>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

lapply

`lapply` always returns a list, regardless of the class of the input.

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.4671
```

```
> x <- list(a = 1:5, b = rnorm(10))
> x
$a
[1] 1 2 3 4 5

$b
 [1] -0.1348413  1.3938458 -1.0369887 -2.1143351  0.7682782
 [6] -0.8161606 -0.4361069  0.9047050 -0.7630863 -0.3410670

> y <- lapply(x,mean)
> y
$a
[1] 3

$b
[1] -0.2575757
```

```
> x <- list(a = 1:5, b = rnorm(10))
> x
$a
[1] 1 2 3 4 5

$b
 [1]  0.5674413 -0.2898360 -0.5120611  0.9778019  0.7808382 -0.7906587
 [7] -0.9630319 -1.7900367  0.9556917  0.6308311

> lapply(x, mean)
$a
[1] 3

$b
[1] -0.04330202
```

Porque o valor de $b é diferente em relação ao exemplo anterior ?

Como controlar os valores gerados por distribuições aleatórias ?

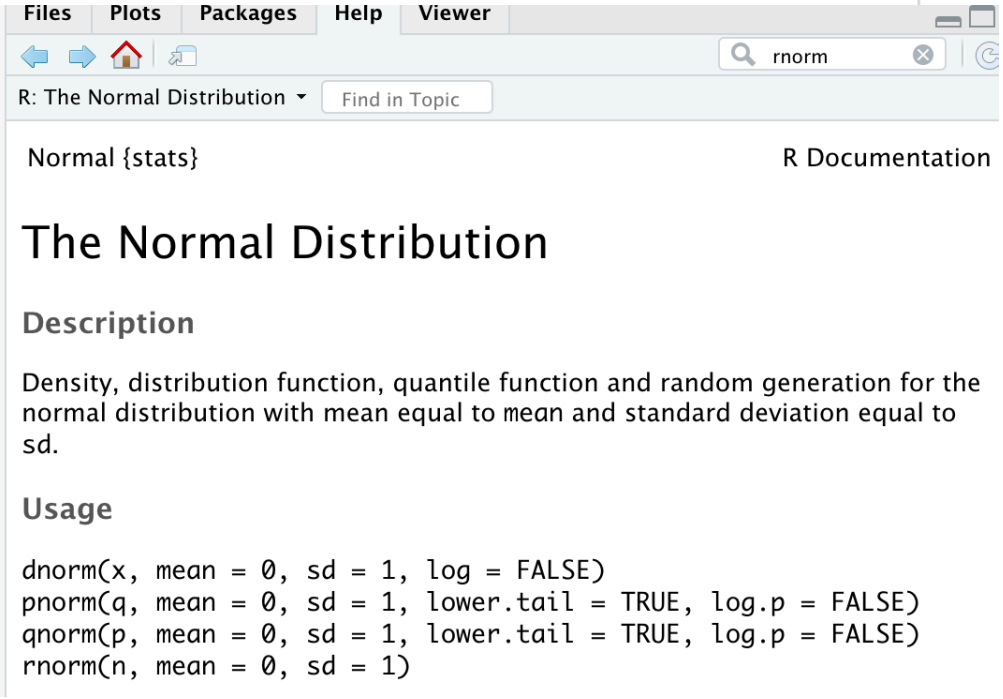
Usando o comando set.seed()

```
> set.seed(10)
> rnorm(10)
 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430
 [7] -1.20807618 -0.36367602 -1.62667268 -0.25647839
> rnorm(10)
 [1]  1.10177950  0.75578151 -0.23823356  0.98744470  0.74139013  0.08934727
 [7] -0.95494386 -0.19515038  0.92552126  0.48297852
> set.seed(10)
> rnorm(10)
 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430
 [7] -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

lapply

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.5261
##
## $c
## [1] 1.421
##
## $d
## [1] 4.927
```



lapply

Trata o vetor como lista

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

```
runif(n, min = 0, max = 1)
```

```
> x <- 1:4
> r <- as.list(x)
> r
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4
```

lapply

Trata o vetor como lista

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

argumentos para a função

sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

sapply

```
> sapply(x, mean)
         a          b          c          d
2.50000000 0.06082667 1.46708277 5.07474950

> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> z <- lapply(x, mean)
> z
$a
[1] 2.5

$b
[1] -0.8162205

$c
[1] 1.115149

$d
[1] 4.766169

> is.list(z)
[1] TRUE
> w <- sapply(x, mean)
> is.list(w)
[1] FALSE
> w
        a         b         c         d
2.5000000 -0.8162205 1.1151485 4.7661694
> is.vector(w)
[1] TRUE
> 
```

Introduction to the R Language

Loop Functions - apply

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

apply

`apply` is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

apply

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

Ex. matriz

- **X** is an array
- **MARGIN** is an integer vector indicating which margins should be "retained".
- **FUN** is a function to be applied
- ... is for other arguments to be passed to **FUN**

Ex. linhas ou colunas

Arrays are the R data objects which can store data in more than two dimensions. For example – If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns.

apply

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```

matriz 20 linhas x 10 colunas

MARGIN = 2 identifica média por coluna

média das 10 colunas

MARGIN = 1 identifica soma por linha

soma das 20 linhas

col/row sums and means

For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are *much* faster, but you won't notice unless you're using a large matrix.

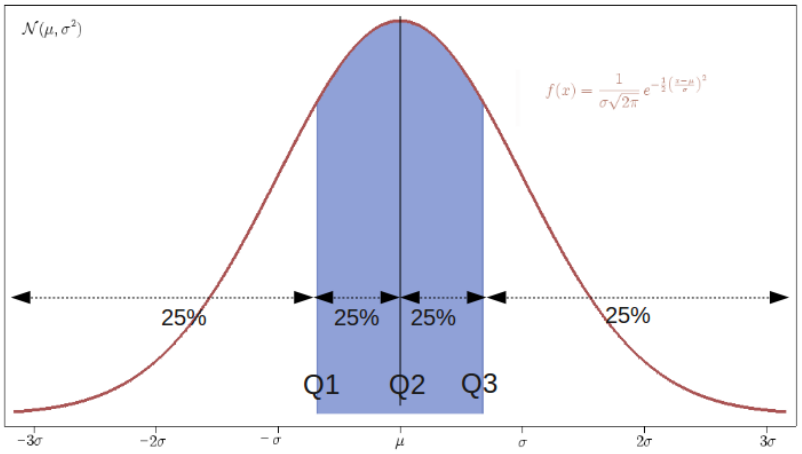
Other Ways to Apply

Quantiles of the rows of a matrix.

Aplica a função quantile a cada uma das linhas

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]        [,2]       [,3]        [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75%  0.9258157  0.07065724  0.3050407 -0.06585436
           [,5]       [,6]      [,7]       [,8]
25% -0.05999553 -0.6588380 -0.653250 0.01749997
75%  0.52928743  0.3727449  1.255089 0.72318419
          [,9]      [,10]      [,11]      [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75%  0.3352377  0.7297176  0.3113434  0.4581150
         [,13]      [,14]      [,15]      [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75%  0.5326299  0.5064267  0.4933852  0.8868922
         [,17]      [,18]      [,19]     [,20]
```

truncado



```
> x <- matrix(rnorm(200), 20, 10)
> x
             [,1]        [,2]         [,3]       [,4]       [,5]
 [1,] -0.694265020 -0.78443232  0.679748875 -0.3006200  0.5416347
 [2,] -0.147266658 -1.05654968  0.474207447 -2.3711774  2.0241848
 [3,] -0.232087413 -0.16202133 -0.681355810  0.8736225  0.6282648
 [4,] -0.001174435  1.41348534 -1.950657504  1.1976114  1.2889882
 [5,]  0.863366242  1.41861703  0.171778325  2.3967907  0.9556607
 [6,]  0.863231890 -1.32951302 -0.518493998  1.0727308  1.2555855
 [7,]  0.012019703  0.32962438  3.199689403  0.7545420  1.0097292
 [8,]  0.023503179  0.92309711  0.719655753  1.3116755 -0.7824640
 [9,]  1.845731769  2.38338943  0.005701253 -1.6307667 -0.2320098
[10,] -0.670594348  0.96682846  0.260494983 -0.6737420  1.5439309
[11,]  0.255385812 -1.96298010  0.908827630  2.5114591  1.6690012
[12,]  0.294112571  0.02163563  0.715514241  0.9388537  0.1557996
[13,]  0.430521260  1.13850313 -1.560414874 -0.9043586  0.7584104
[14,]  0.511416253  0.92696198  1.058149464  0.4648962  1.0733677
[15,] -0.348434963  0.33558167 -0.150499569  0.3140299  0.9994367
[16,]  0.440658004  0.30528549  0.509375547 -0.6769930  0.2440951
[17,] -1.394774512  0.69020738  2.266832633 -1.0825901  1.1203442
[18,]  0.488313918  0.78033589  1.007860593 -2.5265221 -1.0363940
[19,]  1.079452314 -0.25552048 -1.498144880  1.3405280  3.0061669
[20,]  2.082008578 -1.11405769 -1.216608060 -1.5429019 -0.6196631
             [,6]       [,7]        [,8]        [,9]        [,10]
 [1,]  0.29967156  0.3070547  0.43312719  0.32230276  1.608841643
 [2,] -0.57327721 -0.1349026  0.01556131  0.55402299 -0.435533127
 [3,]  1.20105952 -0.4564274  1.08793955  1.11867248  0.361777884
 [4,] -0.34206742 -0.7074910  0.11500646  0.01584398 -0.006004137
 [5,]  0.65025132  0.2961643 -0.12454232  0.01780190  0.978963326
 [6,]  0.54062812  0.8403631 -0.56506458  0.19806431  0.148746540
 [7,] -1.67805958  1.2999364 -1.39097634  0.98390978 -0.679414843
 [8,]  1.95151200 -1.0287042  0.93975538 -1.51741139 -0.697281259
 [9,] -1.19831603  0.8827179 -2.40004764  1.79344402 -0.146585952
[10,]  1.11572221 -1.4876203  1.09485435 -0.08749317  0.691330268
[11,] -0.05740621  0.3659160  0.37167335 -0.43498658  1.370016260
[12,]  1.17697920  2.4090439  0.05242233 -2.05993195 -0.625988774
[13,]  0.59060041  1.9107778  1.05911681  0.49057863 -0.167723646
[14,]  0.85239548  2.0808155 -1.20982923  0.32274302 -0.694489142
[15,] -0.70902008 -0.3238278 -1.24327011  0.40731552 -0.300179483
[16,]  0.08173170 -0.3812877  1.33413735 -1.12468650 -0.293469713
[17,] -1.20515403 -0.2745986  0.02888905  0.97526713  0.381821269
[18,] -0.02829583  0.4329129 -2.52690363 -0.17228406 -0.122850895
[19,] -0.40834907  0.7337814  0.60063263 -0.93320942 -1.097742870
[20,] -0.73256469 -0.4718503 -0.46129465  0.48604520 -0.391892815
```

por linha

```
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]       [,2]       [,3]       [,4]      [,5]       [,6]
25% -0.1505471 -0.5388412 -0.2145709 -0.2580516 0.2028748 -0.3516839
75%  0.5145078  0.3595459  1.0343603  0.9269602 0.9731377  0.8575147
          [,7]       [,8]       [,9]      [,10]     [,11]     [,12]
25% -0.5065562 -0.7611683 -0.9567395 -0.5248191 0.0207918 0.0293323
75%  1.0032743  0.9355908  1.5657625  1.0628479 1.2547191 0.8830188
          [,13]     [,14]      [,15]      [,16]      [,17]      [,18]
25% -0.01816242 0.3582813 -0.3422832 -0.3593332 -0.8805922 -0.8203665
75%  0.98394021 1.0253526  0.3301937  0.4068149  0.9040022  0.4744637
         [,19]      [,20]
25% -0.8019943 -1.0186844
75%  0.9930346 -0.4092433
```

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> a
, , 1

           [,1]       [,2]
[1,]  0.1092981  1.7591204
[2,] -0.2547597 -0.1571301

, , 2

           [,1]      [,2]
[1,] -2.0185152 -2.596028
[2,]  0.3779467  1.124748

, , 3

           [,1]       [,2]
[1,]  1.4317796 -0.3346016
[2,] -0.8773249  0.4985227

, , 4

            [,1]      [,2]
[1,] -0.08852435 0.7648257
[2,] -0.37446574 1.4047519

, , 5

           [,1]        [,2]
[1,] -1.3931792 -0.23722028
[2,] -0.3054796  0.01538012
```

⋮

```
, , 9

            [,1]        [,2]
[1,] -0.81213355  1.58966020
[2,] -0.09228988 -0.04659491

, , 10

           [,1]        [,2]
[1,] -0.4996089 -0.11797526
[2,] -1.4924569  0.09191582
```

apply

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]       [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908

> rowMeans(a, dims = 2)
           [,1]       [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```

MARGIN = c(1, 2) colapsa as linhas e as colunas

dims =2 colapsa o array numa matriz de forma equivalente

Introduction to the R Language

Loop Functions - mapply

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

mapply

`mapply` is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

- `FUN` is a function to apply
- ... contains arguments to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

mapply

The following is tedious to type

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

componente a componente

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

Replicate Elements of Vectors and Lists

Description

`rep` replicates the values in `x`. It is a generic function, and the (internal) default method is described here.

Vectorizing a Function

```
> noise <- function(n, mean, sd) {
+ rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1]  2.4831198  2.4790100  0.4855190 -1.2117759
[5] -0.2743532

> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266  4.2507057  1.1572738
[5]  3.7413584
```

Produz um único resultado

Instant Vectorization

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.658653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

noise(1, 1, 2)

noise(2, 2, 2)

noise(3, 3, 2)

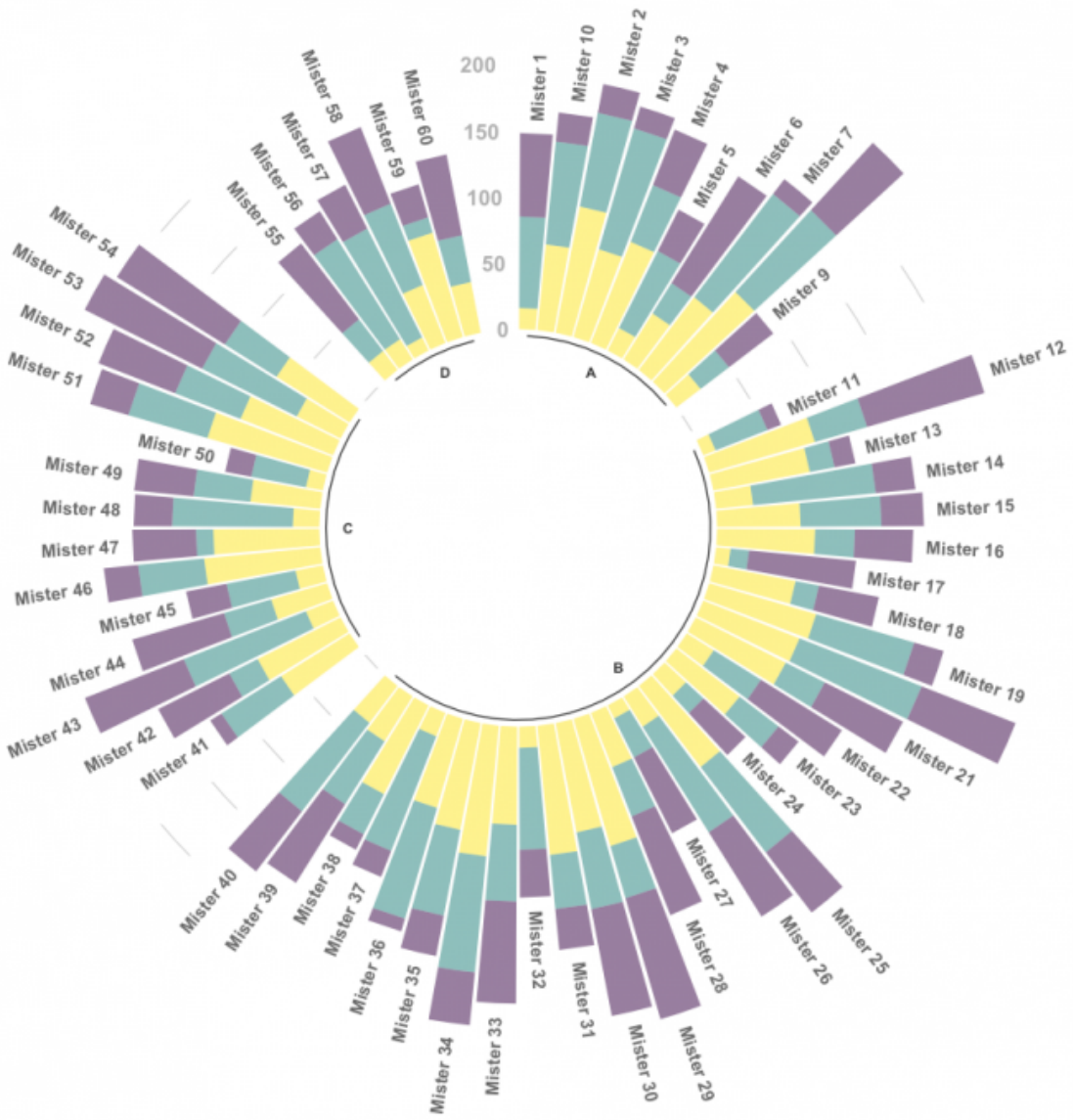
noise(4, 4, 2)

noise(5, 5, 2)

Instant Vectorization

Which is the same as

```
list(noise(1, 1, 2), noise(2, 2, 2),
     noise(3, 3, 2), noise(4, 4, 2),
     noise(5, 5, 2))
```


200
150
100
50
0
A
B
C
D
Mister 1
Mister 10
Mister 2
Mister 3
Mister 4
Mister 5
Mister 6
Mister 7
Mister 9
Mister 11
Mister 12
Mister 13
Mister 14
Mister 15
Mister 16
Mister 17
Mister 18
Mister 19
Mister 21
Mister 22
Mister 23
Mister 24
Mister 25
Mister 26
Mister 27
Mister 28
Mister 29
Mister 30
Mister 31
Mister 32
Mister 33
Mister 34
Mister 35
Mister 36
Mister 37
Mister 38
Mister 39
Mister 40
Mister 41
Mister 42
Mister 43
Mister 44
Mister 45
Mister 46
Mister 47
Mister 48
Mister 49
Mister 50
Mister 51
Mister 52
Mister 53
Mister 54
Mister 55
Mister 56
Mister 57
Mister 58
Mister 59
Mister 60