

```

• begin
•     using ImageFiltering
•     using Images
•     using DSP
• end

```

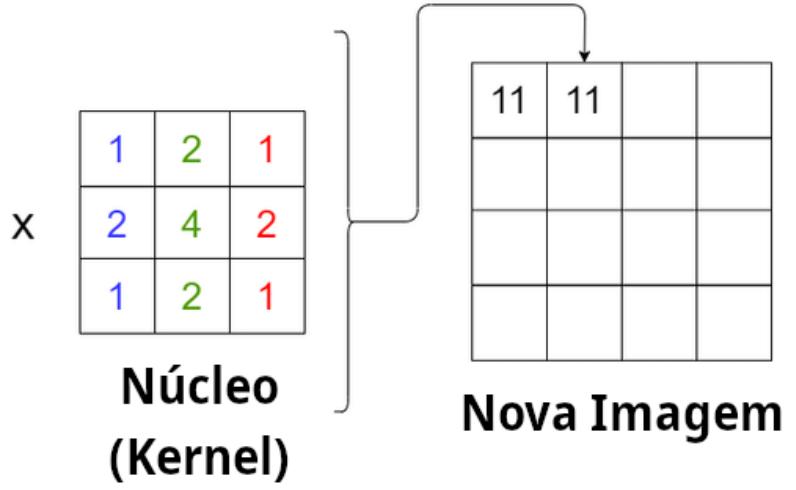
# Introdução à Convoluçãoes

Basicamente, uma convolução, no contexto de processamento de imagens, é um processo onde se forma uma nova imagem a partir de uma operação feita com os pixels vizinhos à um pixel analisado na imagem original

## Exemplo de Convolução

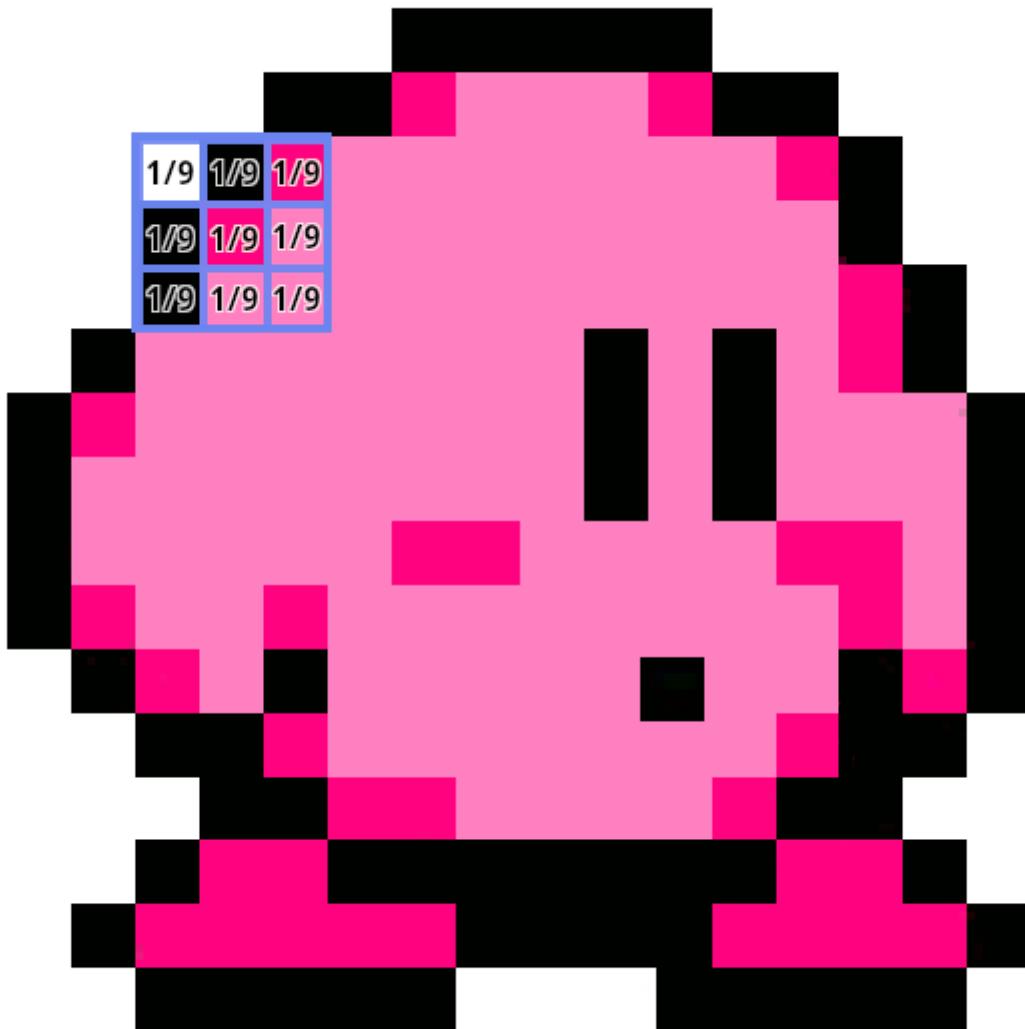
1	1	<sup>*1</sup>	0	<sup>*2</sup>	1	<sup>*1</sup>	0	1
0	1	<sup>*2</sup>	1	<sup>*4</sup>	0	<sup>*2</sup>	1	0
1	0	<sup>*1</sup>	1	<sup>*2</sup>	1	<sup>*1</sup>	0	1
0	1	0	1	1	1	0	0	0
1	0	1	1	1	0	1	1	0
1	0	1	1	1	0	1	1	1

A



## Criando um efeito de borrão

Uma boa ideia para criar um efeito de borrão seria calcular a média dos pixels ao redor do nosso pixel analisado:

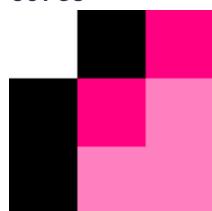


Vamos criar uma função que recebe uma matriz de pixels  $n \times n$  e calcula sua média, retornando o pixel dessa média:

```
media_pixels (generic function with 1 method)
```

```
• function media_pixels(imagem)
•     soma = sum(imagem)
•     n = size(imagem)[1] # Size retornará um valor (n,n), então basta pegar o
      primeiro
•     return soma/n^2
• end
```

```
cores =
```

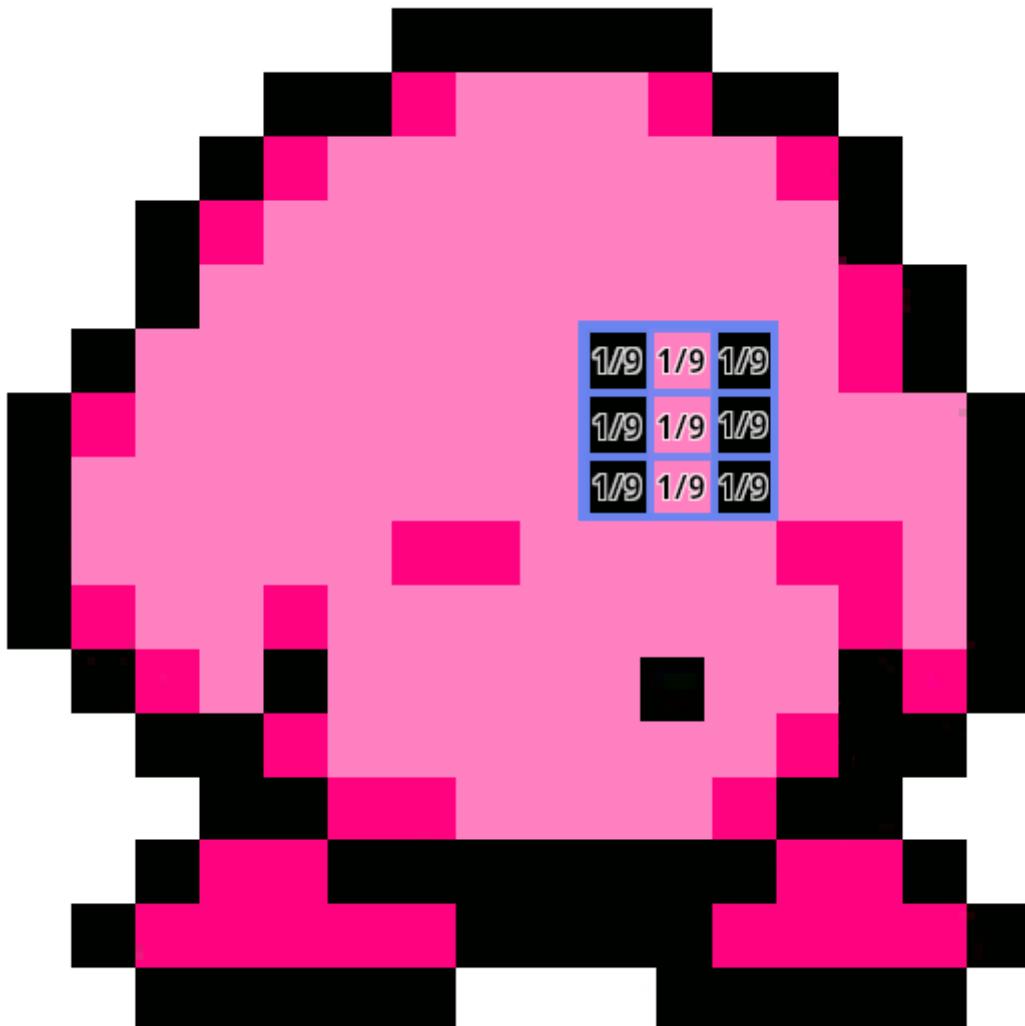


```
• cores = [RGB(1, 1, 1) RGB(0, 0, 0) RGB(1, 0, 0.5)
•           RGB(0, 0, 0) RGB(1, 0, 0.5) RGB(1, 0.5, 0.75)
•           RGB(0, 0, 0) RGB(1, 0.5, 0.75) RGB(1, 0.5, 0.75)]
```

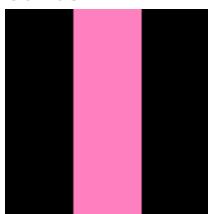


- `media_pixels(cores)`

No contexto de processamento de imagens chamamos essa matriz azul, na qual estamos calculando a média dos pixels ao redor do pixel analisado, de **núcleo** (*kernel*). Consequentemente se movermos o núcleo para outra região da imagem, por exemplo, uma com mais pixels pretos, o pixel médio será mais escuro:



```
cores2 =
```



- `cores2 = [RGB(0, 0, 0) RGB(1, 0.5, 0.75) RGB(0, 0, 0)  
 RGB(0, 0, 0) RGB(1, 0.5, 0.75) RGB(0, 0, 0)]`



- `media_pixels(cores2)`

Aplicando o filtro com a função `imfilter()` (você também pode tentar fazer uma função que aplica nossa função anterior `media_pixels` em toda a imagem como um exercício) obtemos o seguinte resultado:



- `begin`
- `m = fill(1/9,(3,3))`
- `imfilter(load("imgs/kirby32.png"), m)`
- `end`

Para imagens pequenas esse resultado pode ser suficientemente bom, porém, normalmente, quando estamos utilizando núcleos e imagens maiores - por se tratar de uma média - ele acabam sendo um pouco ineficaz. Outra maneira de fazermos esse desfoque é com um **núcleo gaussiano**.

## Efeito de borrão com um núcelo gaussiano

0.002	0.013	0.021	0.013	0.002
0.013	0.059	0.098	0.059	0.013
0.021	0.098	0.162	0.098	0.021
0.013	0.059	0.098	0.059	0.013
0.002	0.013	0.021	0.013	0.002

A imagem acima é uma matriz de uma função gaussiana, caracterizada pela importante **curva normal**, a qual possuí uma "forma de sino". Não é necessário entender seu conceito em sua completude para o resto da aula, apenas a noção de que essa função nos fornece uma matriz mais "equilibrada" do que simplesmente aplicar uma convolução com uma média.

Dentre os pacotes importados um deles é o `ImageFiltering`, o qual possuí uma classe chamada `Kernel` (núcleo), e dentro dessa classe possui-se um método chamado de `gaussian`, ele pode gerar uma matriz com um desvios padrões  $\sigma_1 \dots \sigma_d$  de dimensão  $d$ :

```
nucleo =
5×5 OffsetArray(::Array{Float64,2}, -2:2, -2:2) with eltype Float64 with indices -2:2×-2:
 0.00296902  0.0133062  0.0219382  0.0133062  0.00296902
 0.0133062  0.0596343  0.0983203  0.0596343  0.0133062
 0.0219382  0.0983203  0.162103   0.0983203  0.0219382
 0.0133062  0.0596343  0.0983203  0.0596343  0.0133062
 0.00296902  0.0133062  0.0219382  0.0133062  0.00296902
```

- `nucleo = Kernel.gaussian((1,1)) # Não é necessário definir o parâmetro da dimensão`

Uma observação a ser feita é a de que esse é um `OffsetArray`, por enquanto podemos entendê-lo como um *array* normal só que com um deslocamento em seus índices, no caso, esse se inicia na posição -2 e termina na 2

Vamos aplicar o efeito com uma função do pacote `ImageFiltering` chamada `imfilter()` em uma pintura de M.C.Escher:

```
img =
```





```
• imfilter(img, nucleo)
```

Se nos aumentarmos o tamanho do núcleo o efeito de desfoco fica mais forte:



```
• begin
•     nucleo2 = Kernel.gaussian((5,5))
•     imfilter(img, nucleo2)
• end
```

## Outros exemplos de convoluções

Agora veremos como convoluções podem ser utilizadas de maneiras bem interessantes para obter diversos tipos de tratamento de imagem, considere o seguinte núcleo:

```
nucleo_agucar = 3x3 Array{Float64,2}:
    -0.5  -1.0  -0.5
    -1.0   7.0  -1.0
    -0.5  -1.0  -0.5
• nucleo_agucar = [ -0.5  -1.0  -0.5
•                  -1.0   7.0  -1.0
•                  -0.5  -1.0  -0.5 ]
```



- `imfilter(img, nucleo_agucar)`

Repare que se somarmos todos os elementos do núcleo o resultado será um, isso significa que quando estamos em uma imagem monocromática, ou seja, todos os pixels possuem o mesmo valor, a imagem continuará a mesma, o que mostra como esse núcleo só fará algo não trivial quando há variação nas cores (valores) da imagem, aguçando-os

Outro tipo de convolução que podemos trabalhar é com o efeito *sobel*, ele possui um efeito muito interessante de fazer a detecção de bordas em imagens fazendo algo parecido com o núcleo anterior:

```
nucleo_sobel_v =
3x3 OffsetArray(::Array{Float64,2}, -1:1, -1:1) with eltype Float64 with indices -1:1x-1:
-0.125  -0.25  -0.125
 0.0      0.0    0.0
 0.125   0.25   0.125
```

- `nucleo_sobel_v = Kernel.sobel()[1]` # estaremos especificando o primeiro núcleo pois este método cria dois tipos de detecção, uma horizontal e uma vertical

`v =`

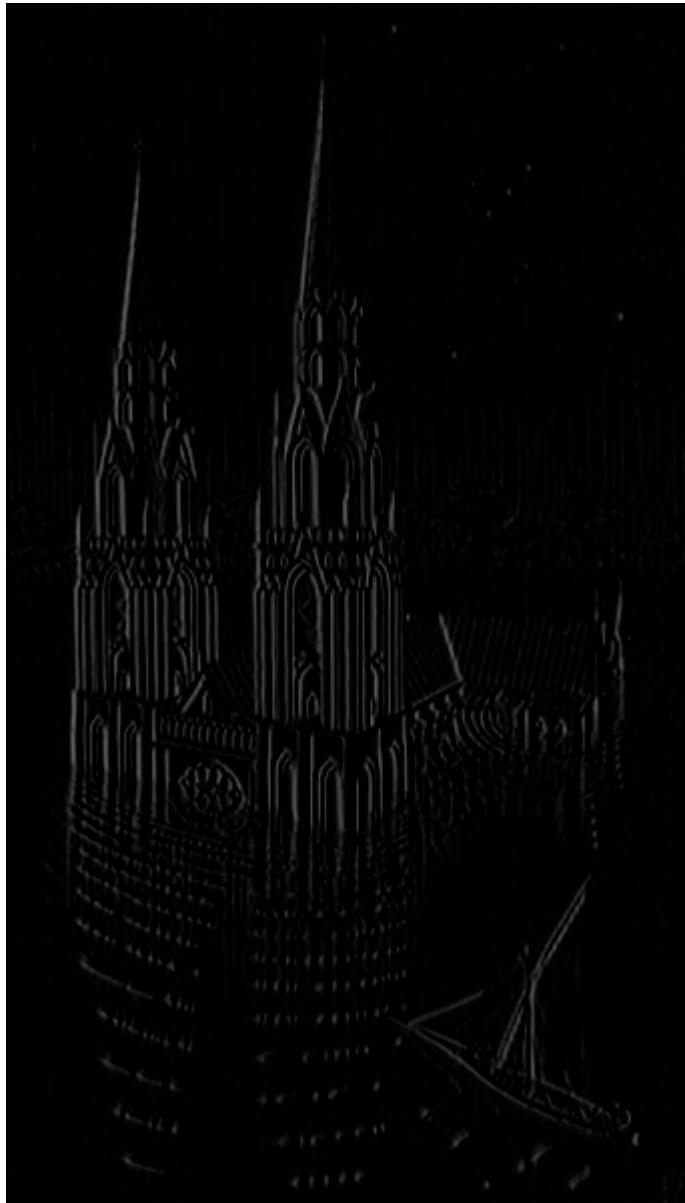


```
• v = imfilter(img, nucleo_sobel_v)
```

```
nucleo_sobel_h =
3x3 OffsetArray(::Array{Float64,2}, -1:1, -1:1) with eltype Float64 with indices -1:1x-1:
-0.125  0.0  0.125
-0.25   0.0  0.25
-0.125  0.0  0.125
```

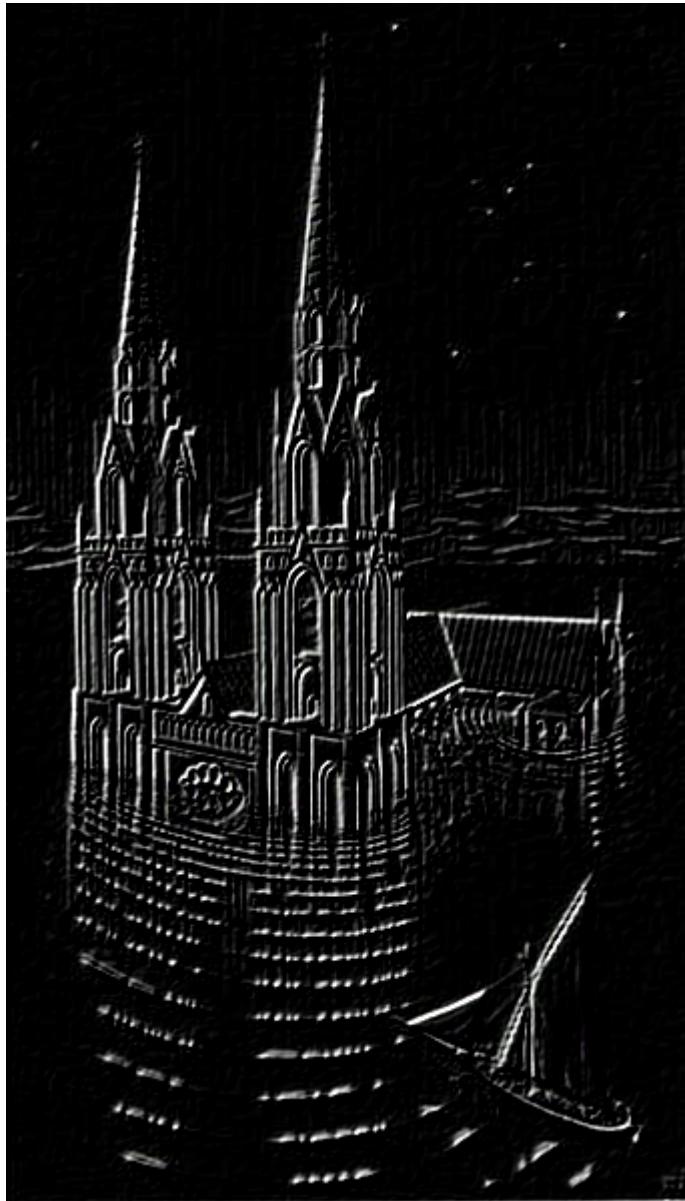
```
• nucleo_sobel_h = Kernel.sobel()[2] # Agora para o filtro horizontal
```

```
h =
```



```
• h = imfilter(img, nucleo_sobel_h)
```

```
total =
```



- `total = (v + h) * 3 # Multiplicamos o valor para ficar mais claro`