

# Algoritmos e Estruturas de Dados

## 1º Projeto

Universidade de Aveiro

Diogo Carvalho, Tiago Garcia



# Algoritmos e Estruturas de Dados

## 1º Projeto

Dept. de Eletrónica, Telecomunicações e Informática  
Universidade de Aveiro

Diogo Carvalho, Tiago Garcia  
(113221) diogo.tav.carvalho@ua.pt, (114184) tiago.rgarcia@ua.pt

22 de novembro de 2023

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>ImageBlur</b>	<b>3</b>
2.1	Análise formal . . . . .	3
2.1.1	Primeira abordagem . . . . .	3
2.1.2	Segunda abordagem . . . . .	4
2.2	Análise experimental . . . . .	4
2.2.1	Primeira abordagem . . . . .	4
2.2.2	Segunda abordagem . . . . .	4
2.3	Algoritmo otimizado da ImageBlur . . . . .	5
2.3.1	Imagem Integral . . . . .	5
2.3.2	Imagem desfocada . . . . .	6
<b>3</b>	<b>Conclusões</b>	<b>8</b>
<b>4</b>	<b>Anexos</b>	<b>9</b>
4.1	ImageBlur: Primeira Abordagem . . . . .	9
4.2	ImageBlur: Segunda Abordagem . . . . .	10

# Capítulo 1

## Introdução

# Capítulo 2

## ImageBlur

### 2.1 Análise formal

#### 2.1.1 Primeira abordagem

##### Descrição do algoritmo

Este algoritmo percorre todos os pixels da imagem original e para cada pixel calcula a média dos pixels num retângulo de desfocagem centrado no pixel atual. Para tal, vai iterar por todos os pixels da imagem e para cada pixel vai iterar de novo por todos os pixels que se encontram no retângulo de dimensões  $2dx + 1 \times 2dy + 1$ , faz a soma dos valores desses pixels, divide pelo número de pixels desse retângulo e atribui esse resultado ao pixel central.

Este algoritmo, por mais que seja simples, fica bastante ineficiente quando o retângulo de desfocagem é grande pois temos de calcular a soma de todos os pixels do retângulo para cada pixel da imagem. Por mais que para imagens pequenas não seja uma diferença notável, à medida que os valores das variáveis aumenta, também o tempo necessário para o algoritmo aumenta imenso.

O código completo deste algoritmo pode ser visto no anexo 4.1.

##### Análise de complexidade

- **Complexidade Temporal**

1. **Desfocagem:** Esta parte da função itera sobre todas as linhas ( $img \rightarrow height$ ) e para cada linha itera sobre todos os pixels da linha ( $img \rightarrow width$ ). Para cada pixel, itera sobre todas as linhas ( $2 * dy + 1$ ) e para cada linha itera sobre todos os pixels da linha ( $2 * dx + 1$ ). Assim, a complexidade temporal desta parte é  $O(img \rightarrow height * img \rightarrow width * (2 * dy + 1) * (2 * dx + 1))$ .

Assim, a complexidade temporal total da função é  $O(img \rightarrow height * img \rightarrow width * (2 * dy + 1) * (2 * dx + 1))$  que significa que a complexidade temporal escala linearmente tanto em função da área da imagem quanto em função da área do retângulo de desfocagem (independentes uma da outra neste requisito).

- **Complexidade Espacial**

1. **Criação da Nova Imagem:** Esta parte da função aloca memória para a nova imagem onde será guardada a imagem desfocada. Uma vez que a nova imagem tem as mesmas dimensões que a imagem original, a complexidade espacial desta parte é  $O(img \rightarrow height * img \rightarrow width)$ .
2. **Desfocagem da Image:** Esta parte modifica a imagem criada, por isso não terá complexidade espacial associada.
3. **Transferência da Nova Imagem:** Esta parte da função apenas transfere o ponteiro da nova imagem para o ponteiro da imagem original, por isso não terá complexidade espacial associada.

Assim, a complexidade espacial total da função é  $O(img \rightarrow height * img \rightarrow width)$  que significa que a complexidade espacial escala linearmente em função da área da imagem.

## 2.1.2 Segunda abordagem

### Descrição do algoritmo

Este algoritmo passa por calcular as somas de todos os pixels num retângulo entre  $(0, 0)$  e  $(x, y)$  antes de começar a criar a imagem desfocada. Uma vez que teremos as somas todas calculadas, podemos calcular o valor de cada pixel da imagem desfocada com base nas somas calculadas anteriormente. Este algoritmo é mais eficiente que o anterior pois não temos de calcular as somas de cada pixel do retângulo de desfocagem, mas sim apenas uma vez para cada pixel da imagem original. Visto que este algoritmo é mais complexo, está explicado mais detalhadamente na secção 2.3.

O código completo deste algoritmo pode ser visto no anexo 4.2.

### Análise de complexidade

#### • Complexidade Temporal

1. **Criação da Imagem Integral:** Esta parte da função itera sobre todas as linhas ( $img \rightarrow height$ ) e para cada linha itera sobre todos os pixels da linha ( $img \rightarrow width$ ). Assim, a complexidade temporal desta parte é  $O(img \rightarrow height * img \rightarrow width)$ .
2. **Desfocagem da Image:** Tal como a anterior, esta parte também itera sobre todas as linhas ( $img \rightarrow height$ ) e para cada linha itera sobre todos os pixels da linha ( $img \rightarrow width$ ). Uma vez que todas as operações dentro dos 'for loops' têm complexidade temporal constante, a complexidade temporal desta parte é  $O(img \rightarrow height * img \rightarrow width)$ .
3. **Limpeza da Imagem Integral:** Esta parte da função tem sempre complexidade temporal constante, por isso a complexidade temporal desta parte é  $O(1)$ .

Assim, a complexidade temporal total da função é  $O(img \rightarrow height * img \rightarrow width) + O(img \rightarrow height * img \rightarrow width) + O(1)$  que simplifica para  $O(img \rightarrow height * img \rightarrow width)$ . Isto significa que a complexidade temporal escala linearmente em função da área da imagem, com esta abordagem, a área do retângulo de desfocagem não irá alterar a complexidade do algoritmo.

#### • Complexidade Espacial

1. **Criação da Imagem Integral:** Esta parte aloca memória para a imagem integral que tem mais uma linha e uma coluna que a imagem original. Assim, a complexidade espacial desta parte é  $O((img \rightarrow height + 1) * (img \rightarrow width + 1))$ .
2. **Desfocagem da Image:** Esta parte modifica a imagem original, por isso não terá complexidade espacial associada.

Assim, a complexidade espacial total da função é  $O((img \rightarrow height + 1) * (img \rightarrow width + 1))$  que simplifica para  $O(img \rightarrow height * img \rightarrow width)$ . Isto significa que a complexidade espacial escala linearmente em função da área da imagem.

## 2.2 Análise experimental

### 2.2.1 Primeira abordagem

### 2.2.2 Segunda abordagem

## 2.3 Algoritmo otimizado da ImageBlur

### 2.3.1 Imagem Integral

Considerando  $img$  como uma matriz  $m \times n$ , onde  $m = img \rightarrow height$  e  $n = img \rightarrow width$ , vamos gerar uma matriz  $m + 1 \times n + 1$  onde a primeira linha e a primeira coluna vão ser zeros, a partir daí, o elemento  $(y, x)$  desta matriz irá corresponder ao elemento  $(y - 1, x - 1)$  da matriz da imagem com  $x \in [1, n]$  e  $y \in [1, m]$ . Esta nova imagem será armazenada como um array de  $m + 1 * n + 1$  elementos, com 32 bits cada (visto que será um array de inteiros).

Para inicializar a matriz, usamos as linhas de código:

```
1  int integral_width = img->width + 1;
2  int integral_height = img->height + 1;
3  int* integral = (int*)calloc(integral_width * integral_height, sizeof(int));
```

Agora vamos ter de completar o resto das células. Cada célula da nova matriz será igual à soma de todos os pixels que estão acima e à esquerda do mesmo.

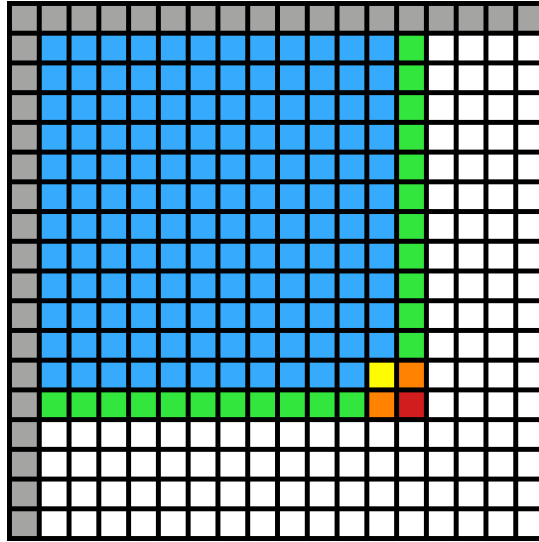


Figura 2.1: Representação da matriz integral

Como podemos ver na figura acima, para calcular o pixel representado a **vermelho** ( $I_{(y,x)}$ ), teremos de adicionar ao pixel original da imagem, depois, o elemento representado acima a **laranja** ( $I_{(y-1,x)}$ ) que já contém todos os pixels acima e à esquerda do mesmo (representados a **verde** acima do pixel mencionado), e por fim, teremos de adicionar o elemento representado à esquerda também a **laranja** ( $I_{(y,x-1)}$ ) que contém todos os pixels na linha horizontal à esquerda do mesmo (representados a **verde** à esquerda do pixel mencionado). Uma vez que o pixel  $I_{(y,x-1)}$ , também contém a área acima dele (parte já incluída pelo pixel  $I_{(y-1,x)}$ ), então teremos de remover a parte duplicada (área representada a **azul**) que corresponde ao valor do pixel representado a **amarelo** ( $I_{(y-1,x-1)}$ ). Com isto, podemos concluir que a expressão para calcular a área a atribuir a um determinado pixel será:

$$I_{(y,x)} = O_{(y-1,x-1)} + I_{(y-1,x)} + I_{(y,x-1)} - I_{(y-1,x-1)} \quad (2.1)$$

De notar que as células representadas a **cinzento** serão as células cujo valor do elemento será sempre zero.

Para calcular todas estas células da matriz, usamos as linhas de código:

```
1  for (int y = 1; y < integral_height; y++) {
2      for (int x = 1; x < integral_width; x++) {
3          integral[y * integral_width + x] = ImageGetPixel(img, x - 1, y - 1)
4              + integral[(y - 1) * integral_width + x]
5              + integral[y * integral_width + (x - 1)]
6              - integral[(y - 1) * integral_width + (x - 1)];
7      }
8  }
```

### 2.3.2 Imagem desfocada

Uma vez que já temos a imagem integral, podemos agora calcular a imagem desfocada. Para isso, vamos percorrer a imagem integral e para cada pixel, teremos de efetuar o seguinte procedimento:

#### 1. Calcular os cantos do retângulo de blur

Vamos calcular a soma dos pixels que estão dentro do retângulo de dimensões  $2dx + 1$  e  $2dy + 1$  para o comprimento e largura, respectivamente, centrado no pixel atual. Poderemos obter esse retângulo usando o canto superior esquerdo bem como o canto inferior direito.

##### (a) Canto superior esquerdo

Para obter cada uma das coordenadas deste ponto, teremos de subtrair a cada coordenada do pixel, os valores do  $dy$  e  $dx$ ,  $dy$  para a borda superior e  $dx$  para a borda da esquerda. No caso do resultado de uma dessas coordenadas ultrapassar um dos limites mínimos do retângulo, então esta coordenada deverá ser o limite mínimo. Este limite normalmente seria 0 mas visto que na imagem integral temos uma linha e uma coluna extra em cima e à esquerda, então temos de somar 1 a esses limites para compensar pelos pixels extra e redefinir o início da parte útil da matriz. Isto fará que o limite mínimo seja 1 à esquerda e em cima. Para calcular as coordenadas, usamos as linhas de código:

```
1 int x1 = x - dx < 1 ? 1 : x - dx;  
2 int y1 = y - dy < 1 ? 1 : y - dy;
```

##### (b) Canto inferior direito

Para obter cada uma das coordenadas deste ponto, teremos de somar a cada coordenada do pixel, os valores do  $dy$  e  $dx$ ,  $dy$  para a borda inferior e  $dx$  para a borda da direita. No caso do resultado de uma dessas coordenadas ultrapassar um dos limites máximos do retângulo, então esta coordenada deverá ser o limite máximo. Uma vez que deste lado não temos nenhuma linha ou coluna extra, o limite máximo será o último índice, ou seja,  $integral\_width - 1$ . Isto fará que o limite máximo seja a largura menos 1 à direita e a altura menos 1 em baixo. Aos resultados, teremos de adicionar o valor 1 visto que não é contabilizada a linha/coluna do pixel central. Para calcular as coordenadas, usamos as linhas de código:

```
1 int x2 = x + dx + 1 > integral_width - 1 ? integral_width - 1 : x + dx + 1;  
2 int y2 = y + dy + 1 > integral_height - 1 ? integral_height - 1 : y + dy + 1;
```

#### 2. Cálculo da quantidade de pixels

Para calcular a quantidade de pixels que estão dentro do retângulo, basta calcular a área do retângulo em pixels, ou seja, a multiplicação do comprimento pela largura. Para tal, usamos a linha de código:

```
1 int count = (x2 - x1) * (y2 - y1);
```



### 3. Cálculo da soma

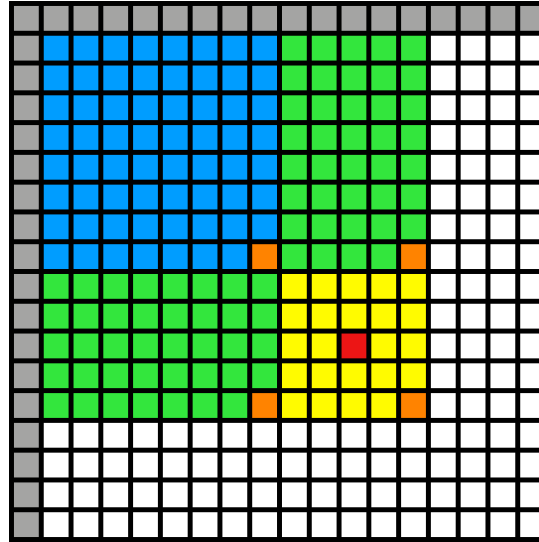


Figura 2.2: Representação do retângulo de blur

Como podemos ver pela figura anterior, para conseguir a área pretendida (representada a **amarelo**), temos de usar como base a área associada ao pixel do canto inferior direito e subtrair a área associada ao pixel do canto superior direito (área acima do retângulo) bem como subtrair a área associada ao pixel do canto inferior esquerdo (área à esquerda do retângulo). Estas áreas exteriores ao retângulo estão representadas a **verde**. Visto que uma parte destas áreas coincide, temos de tirar a parte duplicada que será a área associada ao pixel do canto superior esquerdo e representada a **azul** na imagem. Importante notar que enquanto que o canto inferior direito será incluído na área, os outros 3 cantos não serão. A partir disto, podemos obter a soma do retângulo de blur a partir da seguinte expressão:

$$S_{(y,x)} = I_{(y2,x2)} - I_{(y1,x2)} - I_{(y2,x1)} + I_{(y1,x1)} \quad (2.2)$$

Para calcular o resultado desta expressão, usamos as linhas de código:

```
1  int sum = integral[y2 * integral_width + x2]
2          - integral[y1 * integral_width + x2]
3          - integral[y2 * integral_width + x1]
4          + integral[y1 * integral_width + x1];
```

### 4. Definir o novo valor do pixel

Para definir o novo valor do pixel, basta dividir a soma pelo número de pixels que estão dentro do retângulo. Para tal, usamos a linha de código:

```
1  ImageSetPixel(img, x, y, (sum + count / 2) / count);
```

De notar que somamos  $count/2$  à soma para que o resultado da divisão seja arredondado corretamente. Após repetir este procedimento para todos os pixels da imagem, obtemos a imagem desfocada.

## Capítulo 3

## Conclusões

# Capítulo 4

## Anexos

### 4.1 ImageBlur: Primeira Abordagem

```
1 void ImageBlur(Image img, int dx, int dy) {
2     assert(img != NULL);
3     assert(dx >= 0 && dy >= 0);
4
5     if (dx == 0 && dy == 0) return;
6
7     Image img_new = ImageCreate(img->width, img->height, img->maxval);
8     if (img_new == NULL) return;
9
10    for (int x = 0; x < img->width; x++) {
11        for (int y = 0; y < img->height; y++) {
12            int sum = 0;
13            int count = 0;
14
15            for (int ix = x - dx; ix <= x + dx; ix++) {
16                for (int iy = y - dy; iy <= y + dy; iy++) {
17                    if (!ImageValidPos(img, ix, iy)) continue;
18                    sum += ImageGetPixel(img, ix, iy);
19                    count++;
20                }
21            }
22
23            ImageSetPixel(img_new, x, y, (sum + count / 2) / count);
24        }
25    }
26
27    uint8 *tmp = img->pixel;
28    img->pixel = img_new->pixel;
29    img_new->pixel = tmp;
30
31    ImageDestroy(&img_new);
32 }
```

## 4.2 ImageBlur: Segunda Abordagem

```
1 void ImageBlur(Image img, int dx, int dy) {
2     assert(img != NULL);
3     assert(dx >= 0 && dy >= 0);
4
5     if (dx == 0 && dy == 0) return;
6
7     int integral_width = img->width + 1;
8     int integral_height = img->height + 1;
9     int* integral = (int*)calloc(integral_width * integral_height, sizeof(int));
10    for (int y = 1; y < integral_height; y++) {
11        for (int x = 1; x < integral_width; x++) {
12            integral[y * integral_width + x] = ImageGetPixel(img, x - 1, y - 1)
13                + integral[(y - 1) * integral_width + x]
14                + integral[y * integral_width + (x - 1)]
15                - integral[(y - 1) * integral_width + (x - 1)];
16        }
17    }
18
19    for (int y = 0; y < img->height; y++) {
20        for (int x = 0; x < img->width; x++) {
21            int x1 = x - dx < 1 ? 1 : x - dx;
22            int y1 = y - dy < 1 ? 1 : y - dy;
23
24            int x2 = x + dx + 1 > integral_width - 1 ? integral_width - 1 : x + dx + 1;
25            int y2 = y + dy + 1 > integral_height - 1 ? integral_height - 1 : y + dy + 1;
26
27            int count = (x2 - x1) * (y2 - y1);
28
29            int sum = integral[y2 * integral_width + x2]
30                - integral[y1 * integral_width + x2]
31                - integral[y2 * integral_width + x1]
32                + integral[y1 * integral_width + x1];
33
34            ImageSetPixel(img, x, y, (sum + count / 2) / count);
35        }
36    }
37
38    free(integral);
39 }
```