



Projeto Final – Funções de Processamento de Imagens

1 Introdução

No projeto final da disciplina de Linguagem de Programação, é proposto que o conjunto de técnicas visto durante o curso seja aplicado à área de processamento de imagens.

Desta forma, o aluno deve utilizar o conhecimento adquirido em

- Funções
- Matrizes
- Strings
- Tipos estruturados
- Leitura e escrita de/em arquivos

na solução de alguns problemas básicos envolvendo imagens digitais.

O projeto proposto tem relevância em dois aspectos importantes, a saber, a) o desenvolvimento de técnicas introdutórias de processamento de imagens, que podem ser estendidas posteriormente em trabalhos de pesquisa ou aplicações comerciais e b) o contato com algoritmos presentes em aplicativos populares como Photoshop, Instagram e Facebook, fazendo com que os exemplos didáticos exibidos durante o curso sejam extrapolados para usos em problemas de cunho prático.

2 Critérios de Avaliação

O trabalho a ser entregue deverá ser feito em duplas. Cada dupla deve decidir que tarefa ficará sob responsabilidade de cada membro, de forma que toda a carga de trabalho seja igualmente dividida entre todos.

Tenha em mente que uma parte considerável do trabalho será realizada nas aulas de laboratório e portanto, o professor da disciplina irá observar o andamento das atividades de cada dupla conforme o desenvolvimento do trabalho.

3 Entrega do Trabalho

Os alunos devem apresentar o trabalho desenvolvido em três etapas, conforme explicado a seguir:

- **Etapa 1:** aula de lab. da semana de 04/11
- **Etapa 2:** aula de lab. da semana de 11/11
- **Etapa final:** aula de lab. da semana de 25/11

Na primeira etapa do projeto, a parte básica do sistema de processamento de imagens deverá ser implementada pela dupla durante o horário da aula de laboratório. A parte básica corresponde às funções de abrir imagem, salvar imagem e a uma função de processar imagem que será escolhida no dia da aula de laboratório.

Na segunda etapa do projeto, duas outras funções, escolhidas no dia da aula de laboratório, deverão ser implementadas no horário da aula.

Na etapa final, uma última função será escolhida para implementação. Cada dupla deverá levar o seu projeto pronto para a aula de laboratório. As duplas serão entrevistadas pelo professor para a composição da nota final.

O código do projeto deverá ser enviado ao SIGAA após a conclusão de cada etapa.

Não serão tolerados programas copiados sob nenhuma hipótese: em caso de constatação de cópia, o trabalho dos dois alunos que compõem a dupla será anulado.

4 Imagens Digitais

4.1 Imagem como Matriz

Uma imagem pode ser representada digitalmente como uma matriz na qual cada uma de suas entradas equivale a um pixel (*picture element*). Cada pixel possui um valor associado que vai de 0 (ausência de cor) a 255 (nível máximo de cor).

Imagens monocromáticas (escala de cinza) são formadas por uma única matriz bidimensional, a qual é comumente chamada de canal. Diz-se portanto que uma imagem em escala de cinzas possui um canal. Nestas imagens, o valor em cada posição da matriz denota a intensidade (brilho). A Figura 1 exibe a imagem *peppers.ppm* em níveis de cinza.



Figura 1: *peppers.ppm* em níveis de cinza.

Imagens coloridas são formadas por três canais, que representam as cores vermelho (canal R), verde (canal G) e azul (canal B). Estas são então armazenadas em três matrizes bidimensionais de mesmo tamanho ou equivalente, em uma única matriz de três dimensões. Desta forma, as combinações entre todos os valores possíveis para os canais R, G e B nas entradas da matriz geram as possíveis cores enxergadas pelo olho humano para um determinado pixel. A Figura 2 exibe a imagem *peppers.ppm* original e também os seus canais R, G e B.

4.2 Arquivo de Imagem .ppm

As imagens a serem processadas pelo programa a ser implementado estão no formato de arquivo *.ppm* (*portable pixel map*) em modo texto (ASCII). Este formato de arquivo contém as dimensões da imagem e valores de cada pixel em modo texto, ou seja, ele é legível por humanos (pode ser inspecionado em qualquer editor de texto, como



(a) Imagem original.



(b) Canal R.



(c) Canal G.



(d) Canal B.

Figura 2: peppers.ppm

por exemplo o bloco de notas). Além disso, as funcionalidades vistas na disciplina de leitura e escrita de arquivos funcionam perfeitamente com este formato.

No formato .ppm, cada arquivo de imagem possui a seguinte estrutura:

- Uma string formada por dois caracteres para identificar o tipo do arquivo, seguida por uma quebra de linha. A string é sempre P3, que serve para identificar o formato .ppm utilizado pelo programa.
- O número de colunas (largura), seguido por um espaço em branco, seguido pelo número de linhas da imagem (altura), com uma quebra de linha em seguida.
- O valor máximo admissível por cada pixel (255), seguido por uma quebra de linha.
- Uma quantidade de números entre 0 e 255 igual à quantidade de linhas multiplicada pela quantidade de colunas multiplicada por 3 (número de canais). Estes números denotam os pixels da imagem, dispostos linha por linha: na primeira linha desta parte do arquivo estão todos os pixels da primeira linha da imagem, sendo o primeiro o da primeira coluna, seguidos pelo da segunda coluna e assim por diante. No caso de imagens coloridas, cada pixel tem três valores: primeiro o vermelho (R), seguido pelo verde (G), seguido pelo azul (B).

Como exemplo, a imagem ect_logo.ppm é mostrada na Figura 3 após ter sido redimensionada para 22 linhas e 55 colunas. O conteúdo do seu arquivo é mostrado a seguir. Uma vez que o arquivo é muito grande, são mostrados apenas o R, G e B dos primeiros 4 pixels da primeira e última linha.

```
P3
55 22
255
255 255 255 255 255 254 254 254 85 136 176
...
255 255 255 255 255 255 78 183 213 78 183 213
```

Assim, o primeiro e segundo pixels da primeira linha têm valores (R=255, G=255, B=255), o terceiro tem valores (R=254, G=254, B=254) e o quarto tem valores (R=85, G=136, B=176).

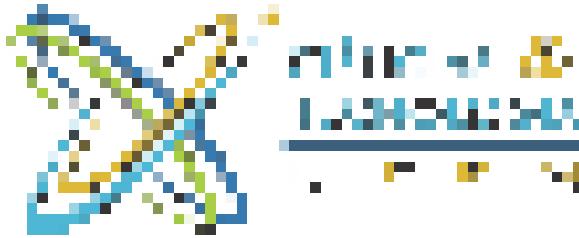


Figura 3: ect_logo.ppm após ter sido redimensionado para o tamanho 22 linhas × 55 colunas (exibido com zoom).

4.3 Tipo Estruturado para Representar Imagens

O projeto deve conter um tipo estruturado definido para armazenar as informações referentes a imagens digitais. Variáveis deste tipo devem ser criadas pelo programa com a finalidade de armazenar os dados a serem lidos/escritos em arquivos .ppm. O tamanho máximo das imagens a serem processadas é de 512×512 (largura × altura).

O tipo estruturado para representar uma imagem deve ter como campos:

- O número de linhas da imagem
- O número de colunas da imagem
- Uma matriz de três dimensões, cujas dimensões são os canais, as linhas e as colunas **ou**
três matrizes de duas dimensões, sendo uma para o canal vermelho, outra para o canal verde e outra para o canal azul **ou**
uma matriz de um tipo estruturado que representa um pixel, sendo este último tipo formado por três valores inteiros denotando as componentes vermelho, verde e azul

5 Implementação do Projeto

5.1 Funções Implementáveis

As funções a serem implementadas realizam operações básicas de algoritmos de processamento de imagens. Todas elas devem operar sobre um tipo estruturado que representa uma imagem. Os algoritmos fornecidos para cada uma destas funções, dados nas seções a seguir, devem ser aplicados para cada um dos canais (vermelho, verde e azul) que compõem uma imagem.

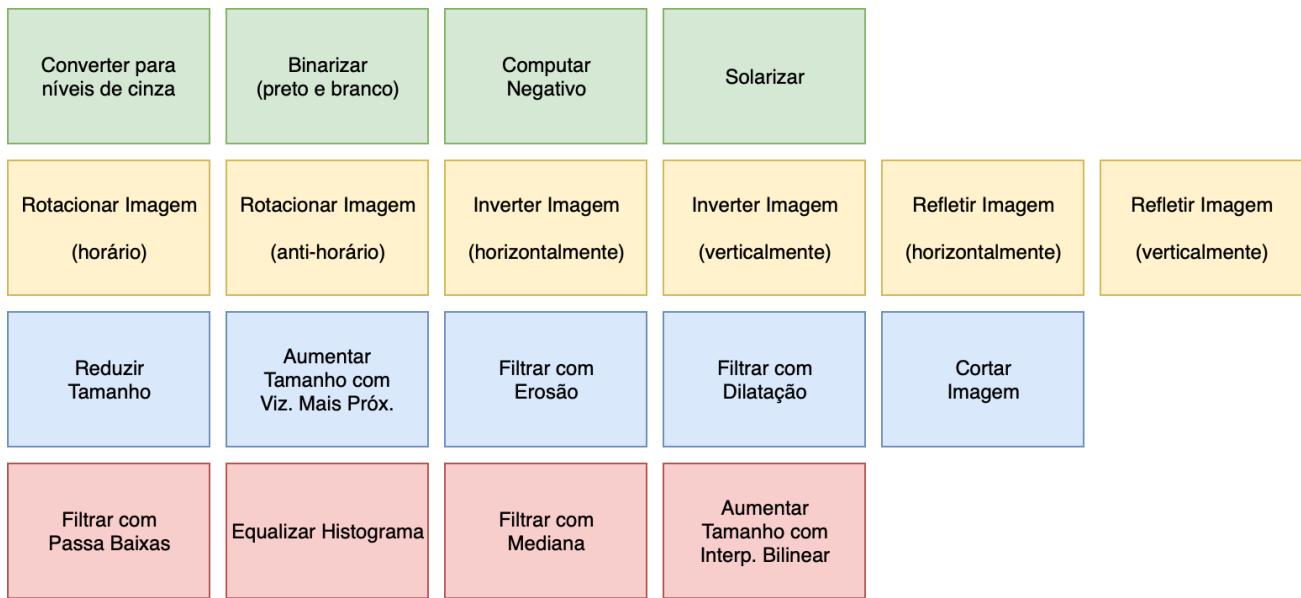


Figura 4: Funções implementáveis no projeto de processamento de imagens.

A Figura 4 exibe cada uma das funções que podem ser sorteadas para implementação nas etapas do projeto. As funções verde, amarela e azul serão sorteadas no início da aula de laboratório de cada subturma. A função vermelha de cada subturma será definida no início da semana de 18/11. Cada etapa do projeto é dada como segue:

- **Etapa 1:** abrir imagem, salvar imagem e uma função do tipo verde
- **Etapa 2:** uma função do tipo amarelo e uma função do tipo azul
- **Etapa final:** menu, checagem de erros e uma função do tipo vermelha

5.2 Menu do Programa

O programa deve possuir um menu em modo texto, de forma que o usuário possa informar qual processamento ele deseja que seja aplicado e então, inserir o nome do arquivo de imagem que deve ser processado. A usabilidade do programa é um dos critérios utilizados para avaliação do trabalho.

À medida em que as imagens resultantes são salvas pelo programa nos arquivos de saída, estas podem ser visualizadas através de qualquer visualizador de imagens compatível com o formato .ppm. O sistema operacional Linux possui visualizador de imagem próprio. Para o Windows, utilize o programa OpenSeeIt¹.

5.3 Checagem de Erros

Considere que cada função deve checar por condições de erro. Por exemplo, ao chamar a função que abre um arquivo para leitura, esta deve checar se o arquivo existe. Esta mesma função também deve checar se o arquivo informado é realmente um arquivo .ppm, já que este é o único formato de imagens aceito pelo programa a ser implementado. A função deve exibir na tela uma mensagem de erro explicando o que houve e encerrar o programa, caso alguma destas situações aconteça. Este mecanismo de checagem de erros vale para todas as funções, cabendo aos desenvolvedores do programa pensar nas possíveis situações de erro e nas mensagens a serem impressas nestas situações.

¹<https://sourceforge.net/projects/openseeit/>

6 Funções a serem Implementadas

6.1 Abrir arquivo de imagem

```
void abre_img(char nome[], Img& img);
```

Esta função deve receber como parâmetro de entrada uma string contendo o nome do arquivo de imagem e um parâmetro de saída do tipo imagem. No parâmetro de saída, a função deve armazenar todos os pixels lidos do arquivo .ppm informado. O nome do arquivo a ser aberto deverá ser informado pelo usuário no menu principal (função `main`).

Nenhuma função a não ser a função `main` deve realizar chamadas a esta função, respeitando desta forma que cada função realize somente aquilo que ela deve fazer. Em outras palavras, funções de processamento não devem salvar imagens. Também, você pode considerar a função retornar um `bool` para que ela informe se houve algum erro de processamento (por exemplo, o arquivo não pôde ser aberto ou o arquivo não é uma imagem válida).

6.2 Salvar arquivo de imagem

```
void salva_img(char nome[], Img img);
```

Esta função deve receber como parâmetros de entrada uma string contendo o nome do arquivo de imagem e uma variável do tipo imagem. A função deve abrir um arquivo com o nome informado e nele salvar todos os valores presentes na variável do tipo imagem passada como parâmetro.

A implementação de como o nome do arquivo será informado à função fica a cargo de cada grupo. A única restrição é que o arquivo a ser salvo não pode ser o mesmo arquivo de entrada.

Nenhuma função a não ser a função `main` deve realizar chamadas a esta função. Também, você pode considerar a função retornar um `bool` para que ela informe se houve algum erro de processamento (por exemplo, o arquivo não pôde ser salvo).

6.3 Conversão para níveis de cinza

```
void converte_para_cinza(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve converter cada pixel colorido no seu equivalente em níveis de cinza. Para isto, computa-se a média entre os valores R, G e B de cada pixel. Observe que uma vez que a imagem de saída tem cada pixel representado por apenas um valor, este último deve ser replicado em cada valor R, G e B. Por exemplo, caso a média do pixel na posição i, j seja igual a V , os valores do canal R, G e B da posição i, j serão iguais a V . Esta conversão pode ser vista na Figura 1.

6.4 Binarização de imagem (preto e branco)

```
void binariza(Img img_in, Img& img_out, int limiar);
```

Esta função deve receber como parâmetros de entrada uma imagem e um número inteiro, além de uma outra imagem como parâmetro de saída. A função deve converter cada pixel em um pixel binário (cujo valor é 0 ou 255), de acordo com um limiar dado pelo número inteiro: todo pixel que possuir valor em nível de cinza (obtido pela média entre os valores dos seus canais R, G e B) inferior ao limiar deve ser convertido em 0 e todo pixel que possuir valor em nível de cinza maior ou igual ao limiar deve ser convertido em 255. O resultado da operação pode ser visualizado na Figura 5.



Figura 5: Binarização para preto e branco em lena.ppm

6.5 Negativo de imagem

```
void computa_negativo(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve computar o negativo de uma imagem: para cada canal R, G e B, o negativo de um pixel é dado por 255 (valor máximo) menos o seu valor. O resultado da operação pode ser visualizado na Figura 6.

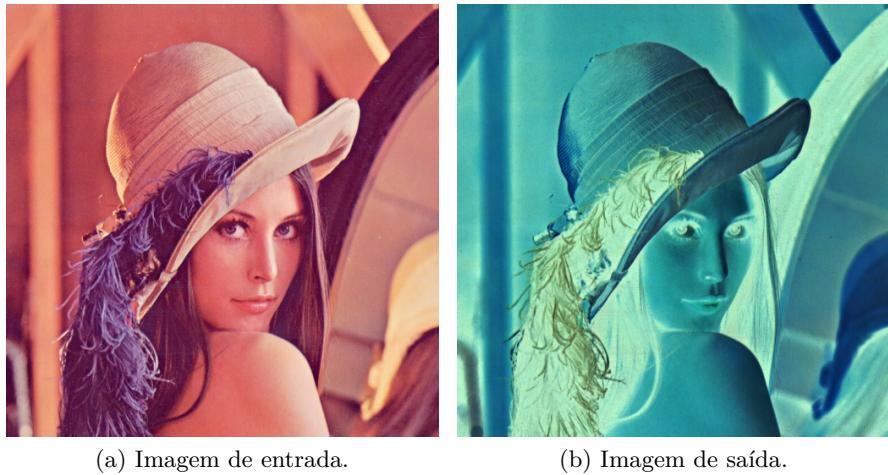


Figura 6: Negativo de lena.ppm

6.6 Solarização de imagem

```
void solariza(Img img_in, Img& img_out, int limiar);
```

Esta função deve receber como parâmetros de entrada uma imagem e um número inteiro, além de uma outra imagem como parâmetro de saída. A função deve solarizar cada pixel, de acordo com um limiar dado pelo número inteiro: em cada canal independente R, G e B, todo pixel que possuir valor inferior ao limiar deve ser convertido no

seu negativo (de acordo com a função anterior); caso contrário, o seu valor original deve ser mantido. O resultado da operação pode ser visualizado na Figura 7.

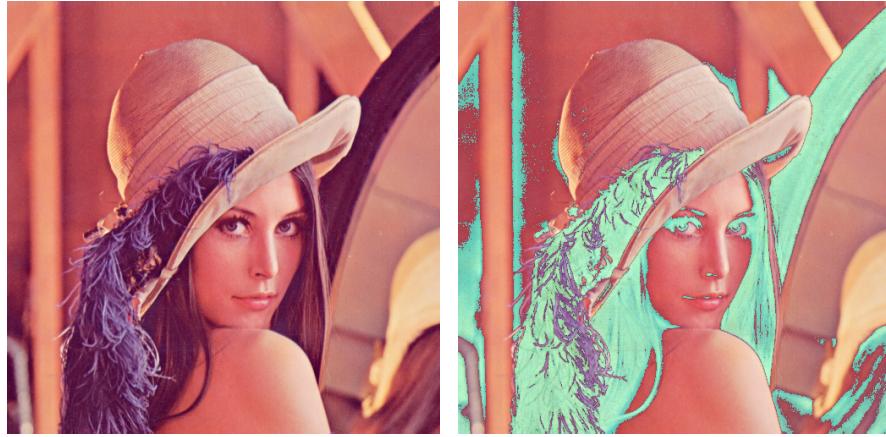


Figura 7: Solarização de lena.ppm

6.7 Rotação de imagem em 90° no sentido horário/anti-horário

```
void rotaciona(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar uma rotação de 90° na imagem de entrada, podendo esta rotação ser no sentido horário ou anti-horário. O resultado deve ser armazenado no parâmetro de saída, e pode ser visualizado na Figura 8.



Figura 8: Rotação no sentido horário em lena.ppm

6.8 Inversão horizontal/vertical na imagem (*flip*)

```
void inverte(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. Duas versões para esta função podem ser implementadas: inversão horizontal ou inversão vertical. Na primeira versão, a função deve realizar uma inversão nos valores dos pixels da imagem de entrada no sentido horizontal (ou seja, em torno de um eixo vertical), armazenando o resultado no parâmetro de saída. Na segunda versão, a função inverte a imagem de entrada no sentido vertical (ou seja, em torno de um eixo horizontal). O resultado para as duas versões pode ser visualizado na Figura 9.



(a) lena.ppm invertida horizontalmente. (b) lena.ppm invertida verticalmente.

Figura 9: Inversão horizontal e vertical em lena.ppm

6.9 Reflexão horizontal/vetical na imagem (*mirror*)

```
void reflete(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. Duas versões para esta função podem ser implementadas: reflexão horizontal ou reflexão vertical. Na primeira versão, a função deve realizar uma reflexão nos valores dos pixels da imagem de entrada em torno de um eixo vertical localizado no centro da imagem (reflexão horizontal), armazenando o resultado no parâmetro de saída. Na segunda versão, a função reflete a imagem de entrada em torno de um eixo horizontal também no centro da imagem (reflexão vertical). O resultado para as duas versões pode ser visualizado na Figura 10.

Bônus: implementar a função com um parâmetro de entrada a mais dado por um número inteiro que corresponde ao índice da coluna/linha que será utilizada como eixo de reflexão.



(a) lena.ppm refletida horizontalmente. (b) lena.ppm refletida verticalmente.

Figura 10: Reflexão horizontal e vertical em lena.ppm

6.10 Redução de tamanho da imagem

```
void diminui_tamanho(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve reduzir o tamanho da imagem de entrada, como ilustrado na Figura 11.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(0,0)	(0,2)	(0,4)	(0,6)
(2,0)	(2,2)	(2,4)	(2,6)
(4,0)	(4,2)	(4,4)	(4,6)
(6,0)	(6,2)	(6,4)	(6,6)

(a) Matriz da imagem original.

(b) Matriz da imagem original, com entradas a serem descartadas.

(c) Matriz final.

Figura 11: Intuição do algoritmo para diminuir o tamanho de uma imagem. As entradas da matriz em azul são iguais às da imagem original e as em vermelho são as que devem ser descartadas. Observe os índices da imagem original que compõem a imagem final.

Considerando uma imagem de entrada com M colunas e N linhas (Figura 11a), o algoritmo deve descartar todas as suas linhas e colunas ímpares (Figura 11b) e agrupar os pixels resultantes na imagem final (Figura 11c), que possui tamanho $M/2 \times N/2$. Um exemplo de resultado para a redução de tamanho de uma imagem pode ser visto na Figura 12.



(a) Imagem de entrada. (b) Imagem de saída.

Figura 12: Redução de tamanho em lena.ppm

6.11 Aumento de tamanho da imagem com vizinhos mais próximos

```
void aumenta_tamanho_vmp(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve aumentar o tamanho da imagem de entrada utilizando o algoritmo de vizinhos mais próximos, como ilustrado na Figura 13.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(a) Matriz da imagem original.

0,0	0,0	0,1	0,1	0,2	0,2	0,3
0,0	0,0	0,1	0,1	0,2	0,2	0,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3
1,0	1,0	1,1	1,1	1,2	1,2	1,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3
2,0	2,0	2,1	2,1	2,2	2,2	2,3
3,0	3,0	3,1	3,1	3,2	3,2	3,3

(b) Matriz da imagem final, com entradas a serem copiadas da imagem original.

Figura 13: Intuição do algoritmo para aumentar o tamanho de uma imagem utilizando o algoritmo de vizinhos mais próximos. As entradas da matriz em amarelo devem ser copiadas da posição indicada na imagem original, replicando assim um pixel nos seus vizinhos mais próximos.

Considerando uma imagem de entrada com M colunas e N linhas (Figura 13a), o algoritmo deve gerar uma imagem de $2M - 1$ colunas e $2N - 1$ linhas. Para isto, o algoritmo deve copiar o valor do pixel de posição (i, j) nas posições $(i + 1, j)$, $(i, j + 1)$ e $(i + 1, j + 1)$, isto é, nos seus pixels vizinhos. Observe que estas posições podem estar fora da imagem e por isso, o algoritmo deve checar os limites e não realizar a cópia se for o caso. Um exemplo de

resultado para o aumento de tamanho de uma imagem com o algoritmo de vizinhos mais próximos pode ser visto na Figura 14.

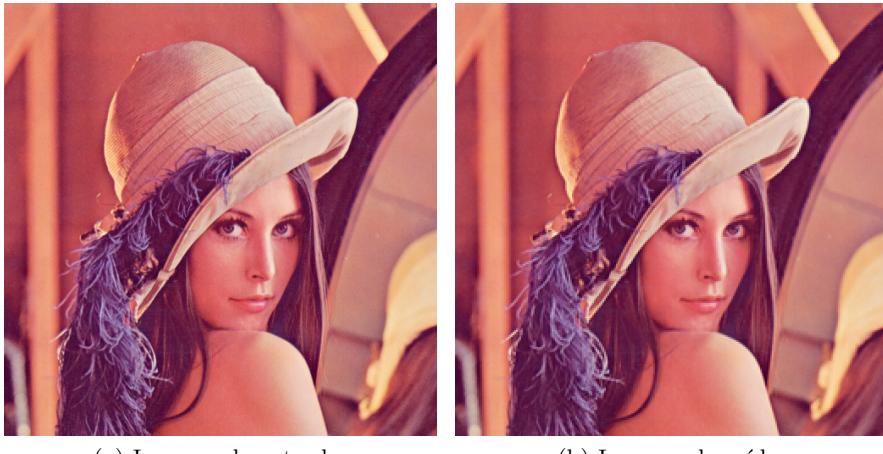


Figura 14: Aumento de tamanho com vizinhos mais próximos em lena.ppm

6.12 Filtragem por erosão

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar a operação de filtragem por erosão em uma imagem binarizada (com valores 0 ou 255), o que é feito com um filtro de tamanho 2×2 como mostra a Figura 15a. Os coeficientes do filtro são formados por valores lógicos (0 ou 1).

Para realizar esta operação, a origem do filtro (canto superior esquerdo ou posição 0,0 da matriz 2×2 , mostrada com um círculo azul na Figura 15a) deve ser posicionada sobre todos os pixels da imagem original, com exceção dos pixels da última coluna e da última linha. Supondo que a origem encontra-se sobreposta à posição i, j da imagem original F , o valor na mesma posição i, j da imagem resultante somente será igual a 255 caso todos os valores $F(i, j)$, $F(i + 1, j)$, $F(i, j + 1)$ e $F(i + 1, j + 1)$ sejam iguais a 255. Caso contrário, o pixel resultante na posição i, j será igual a 0.

O processo é exibido em uma imagem reduzida na Figura 15, com um resultado em uma imagem real mostrado na Figura 16.

Observe que a operação só é válida para uma imagem binária. Portanto, existem duas possibilidades de operação para a função: ela pode receber como entrada uma imagem já binarizada previamente ou ela pode receber uma imagem qualquer e realizar uma chamada à função de binarização como passo intermediário, armazenando o resultado em uma variável local da função.

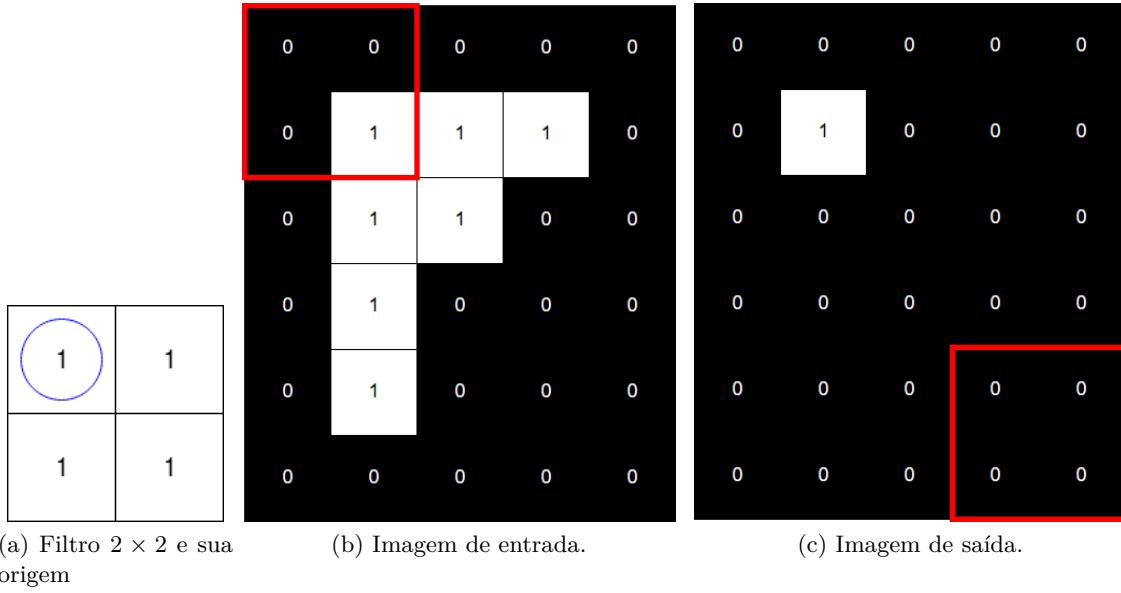


Figura 15: Exemplo da operação de erosão. Em vermelho, estão a) a primeira posição em que o filtro deve ser posicionado e b) a última posição em que o filtro deve ser posicionado.

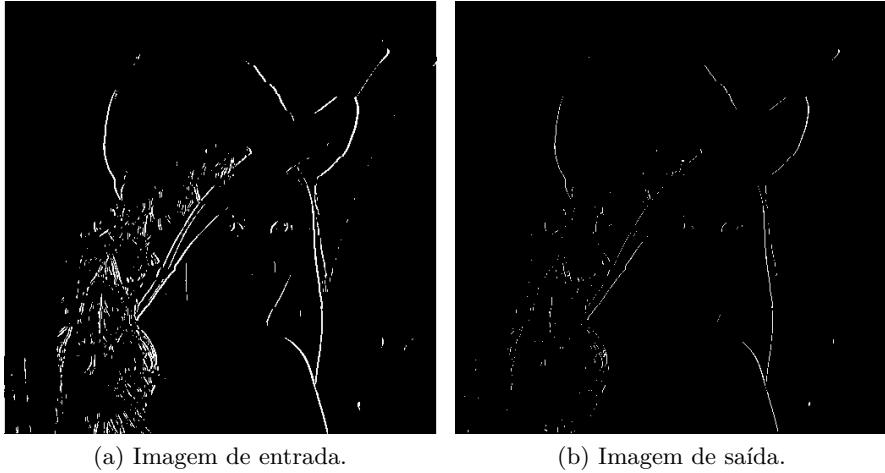


Figura 16: Filtragem por erosão em lena_cont.ppm

6.13 Filtragem por dilatação

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar a operação de filtragem por dilatação em uma imagem binarizada (com valores 0 ou 255), o que é feito com um filtro de tamanho 2×2 como mostra a Figura 17a. Os coeficientes do filtro são formados por valores lógicos (0 ou 1).

Para realizar esta operação, a origem do filtro (canto superior esquerdo ou posição 0, 0 da matriz 2×2 , mostrada com um círculo azul na Figura 17a) deve ser posicionada sobre todos os pixels da imagem original, com exceção dos pixels da última coluna e da última linha. Supondo que a origem encontra-se sobreposta à posição i, j da imagem original F , o valor na mesma posição i, j da imagem resultante será igual a 255 caso pelo menos um valor

$F(i, j)$, $F(i + 1, j)$, $F(i, j + 1)$ ou $F(i + 1, j + 1)$ seja igual a 255. Caso contrário, o pixel resultante na posição i, j será igual a 0.

O processo é exibido em uma imagem reduzida na Figura 17, com um resultado em uma imagem real mostrado na Figura 18.

Observe que a operação só é válida para uma imagem binária. Portanto, existem duas possibilidades de operação para a função: ela pode receber como entrada uma imagem já binarizada previamente ou ela pode receber uma imagem qualquer e realizar uma chamada à função de binarização como passo intermediário, armazenando o resultado em uma variável local da função.

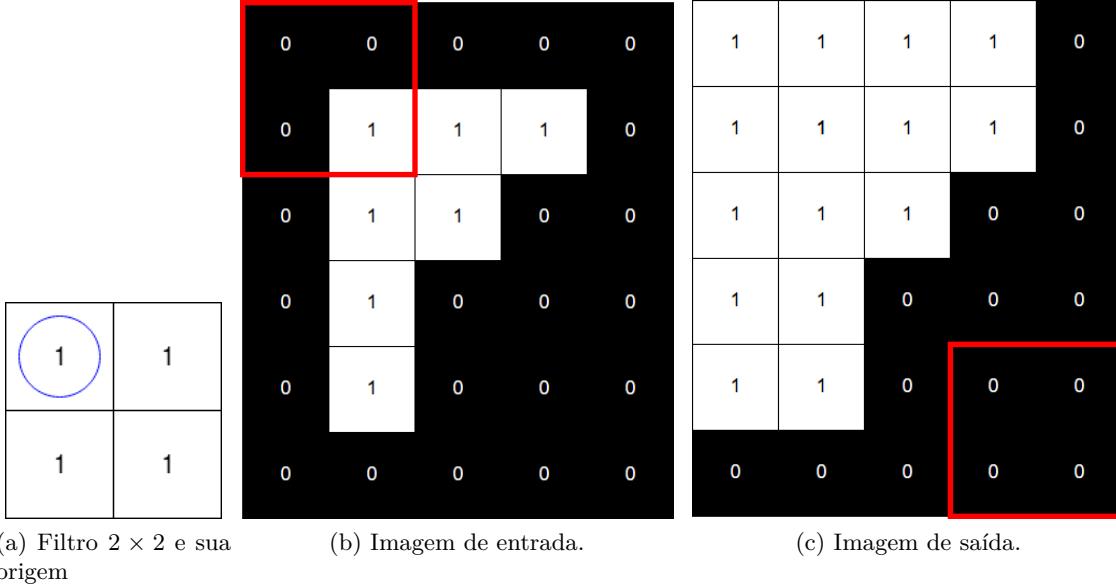


Figura 17: Exemplo da operação de dilatação. Em vermelho, estão a) a primeira posição em que o filtro deve ser posicionado e b) a última posição em que o filtro deve ser posicionado.

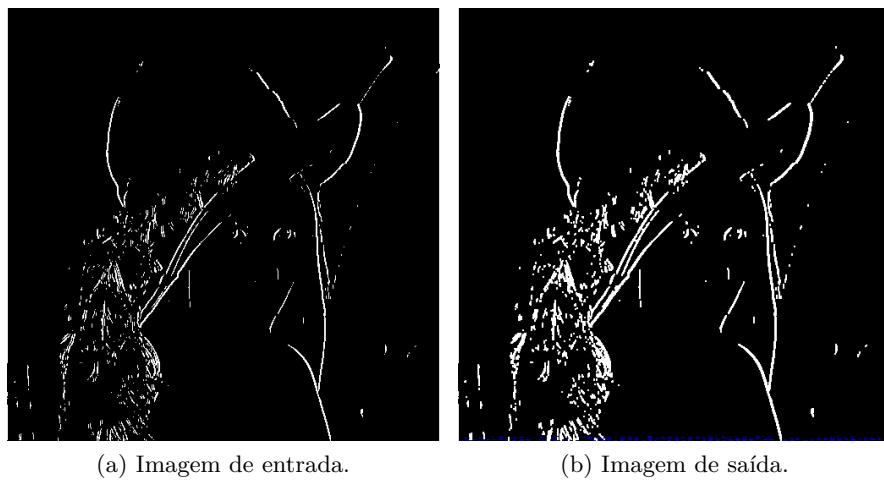


Figura 18: Filtragem por dilatação em lena_cont.ppm

6.14 Recorte de subimagem (*crop*)

```
void corta_rgb(Img img_in, Img& img_out, int li, int ci, int larg, int alt); ou  
void corta_rgb(Img img_in, Img& img_out, int li, int ci, int lf, int cf);
```

Esta função deve receber como parâmetros de entrada a imagem a ser processada e também valores inteiros informando a região de corte: linha e coluna iniciais seguidas pela largura e altura do retângulo. Alternativamente, a região de corte também pode ser delimitada pela linha e coluna iniciais seguidas pela linha e coluna finais. A subimagem correspondente à região de corte deve ser armazenada em uma imagem passada como parâmetro de saída.

Quando esta função for chamada pelo programa, os valores para os parâmetros de corte devem ser gerados aleatoriamente. Considerando uma imagem de entrada com M colunas e N linhas, os parâmetros de corte devem ser inteiros no intervalo $[0, M - 1]$ para as colunas e $[0, N - 1]$ para as linhas.

A Figura 19 exibe a imagem lena.ppm e uma subimagem correspondente à linha e coluna inicial iguais a 0, com largura e altura iguais a 128.



Figura 19: Crop em lena.ppm

6.15 Convolução para filtragem passa baixas

```
void borra_imagem(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar a operação de convolução da imagem de entrada com um filtro, com o objetivo de realizar o borramento da imagem (filtragem passa baixas). O processo é ilustrado na Figura 20.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

(a) Filtro borrador e cada um dos seus coeficientes.

				2	0	5		
			1	100	100			
		100	100	100	0			

(b) Cálculo do pixel de saída da terceira linha e quinta coluna.

Figura 20: Operação de convolução para borramento de uma imagem.

Um filtro é uma matriz de dimensão bem menor em comparação com o tamanho de imagens convencionais. Normalmente, se usam filtros de tamanho 3×3 , 5×5 ou 7×7 .

A operação de convolução calcula o valor de um pixel na imagem de saída pela soma ponderada dos valores vizinhos ao pixel da imagem de entrada. O filtro é sobreposto na posição correspondente da imagem de entrada, fazendo com que a soma seja ponderada de acordo com as entradas da matriz (coeficientes) do filtro. Matematicamente, um pixel $G(i, j)$ da imagem de saída é dado pela Equação 1, na qual H é um filtro de tamanho 3×3 e F é a imagem de entrada.

$$G(i, j) = \sum_{p=0}^2 \sum_{q=0}^2 F(i + p - 1, j + q - 1)H(p, q) \quad (1)$$

A Figura 20b exibe a intuição do processo de convolução. Observe que o valor do pixel de saída sendo calculado é sempre sobreposto com o coeficiente central do filtro. Neste exemplo, o valor para o pixel $G(2, 4)$, situado na terceira linha e quinta coluna da imagem de saída, é igual a

$$\begin{aligned} G(2, 4) &= (1/9) \times 2 + (1/9) \times 0 + (1/9) \times 5 + \\ &= (1/9) \times 1 + (1/9) \times 100 + (1/9) \times 100 + \\ &= (1/9) \times 100 + (1/9) \times 100 + (1/9) \times 0 \\ &= 408/9 = 45. \end{aligned}$$

Assim, para se realizar a convolução da imagem com um filtro basta computar a Equação 1 para todos os pixels da imagem de saída. Note que, ao computar os valores para pixels nas bordas da imagem, os coeficientes do filtro irão sobrepor pixels fora dos limites da imagem (para uma imagem de M colunas e N linhas, pixels em linha/coluna igual a -1 ou linha igual a N ou ainda em coluna igual a M). O algoritmo a ser implementado deve considerar valores iguais a 0 para estes pixels. Um exemplo de resultado para o borramento de uma imagem pode ser visto na Figura 21.

O algoritmo de borramento da imagem pode ser implementado com o filtro tendo tamanho fixo igual a 3×3 ou com tamanho parametrizável. Neste último caso, o tamanho do filtro $S \times S$ pode ser dado por um número ímpar S , que deve ser parâmetro da função e deve ser lido do usuário no menu do programa.



Figura 21: Borramento em lena.ppm

6.16 Equalização de histograma

```
void equaliza_histograma(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar a operação de equalização de histograma da imagem de entrada. Este processo é realizado em duas etapas.

Em uma primeira etapa, o algoritmo deve computar o histograma da imagem de entrada, isto é, quantas vezes cada valor de pixel válido ($0, 1, 2, 3, \dots, 255$) aparece na imagem de entrada. Matematicamente, sendo r um valor de pixel na imagem de entrada, a primeira etapa computa um histograma $h(r)$, dado por um vetor de contadores de 256 posições cuja posição r armazena quantas vezes r ocorre na imagem.

Na segunda etapa, o valor r presente em cada pixel da imagem de entrada deve ser substituído por s , o valor a ser inserido na imagem de saída na mesma posição do pixel original. O valor s é dado pela Equação 2.

$$s = \frac{255}{M \times N} \sum_{k=0}^r h(k) \quad (2)$$

Assim, para um valor igual a 15 na imagem original, o valor na imagem de saída deve ser igual ao somatório de quantas vezes ocorre o valor 0, somado com quantas vezes ocorre o valor 1, somado com todos os valores até quantas vezes ocorre o valor 15, multiplicado por 255 e dividido pelo total de pixels da imagem de entrada. Observe que, como as imagens possuem três canais, é necessário computar um histograma para cada cor: h_r (vermelho), h_g (verde) e h_b (azul).

Um exemplo de resultado para a equalização de histograma de uma imagem pode ser visto na Figura 22.



(a) Imagem de entrada.

(b) Imagem de saída.

Figura 22: Equalização de histograma em lena.ppm

6.17 Remoção de ruído com filtro da mediana

```
void remove_ruido_mediana(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. A função deve realizar a operação de filtragem com o filtro da mediana na imagem de entrada. Este tipo de filtro é utilizado para remover ruídos em imagens do tipo “sal e pimenta”, o qual corrompe imagens com ruído que alteram pixels aleatoriamente em seus valores máximos (255 ou sal) ou mínimos (0 ou pimenta).

A função deve realizar a operação de filtragem com o filtro da mediana na imagem “sal e pimenta”. Para filtrar uma imagem com o filtro da mediana, deve-se utilizar uma janela deslizante de tamanho 3×3 similar ao filtro da Seção 6.15. O ponto central da janela deve ser sobreposto a todos os pixels válidos da imagem “sal e pimenta”, afim de computar o valor na imagem de saída. Todos os 9 pixels que estão dentro da janela são colocados em um vetor e então ordenados em ordem crescente de valor. O valor do pixel na imagem de saída é igual ao pixel que encontra-se exatamente no meio do vetor. Este valor é a mediana.

Considerando o exemplo na Figura 20b, o valor $G(2, 4)$ da imagem de saída G é obtido ordenando-se o vetor com todos os valores dentro da janela de $F(2, 4)$. O vetor ordenado é igual a $V = [0, 0, 1, 2, 5, 100, 100, 100, 100]$ e o valor do pixel na imagem de saída, que deve ser igual ao valor situado na posição do meio do vetor, deve ser portanto igual a 5.

Uma vez que a imagem a ser processada pela função deve estar contaminda por ruído sal e pimenta, o seu programa deve implementar uma função para corromper a imagem de entrada com ruído, de acordo com uma porcentagem informada em um outro parâmetro de entrada.

O parâmetro de entrada contendo a porcentagem será utilizado para definir a taxa percentual de pixels da imagem original que serão contaminados. Por exemplo, se a imagem de entrada tem tamanho 100×100 pixels e o parâmetro porcentagem é igual a 25, temos que 2500 pixels (25% do total de 10000) deverão ser contaminados por ruído. Deste total, 1250 pixels escolhidos aleatoriamente serão modificados com valores máximos (255 ou sal) e 1250 pixels também escolhidos aleatoriamente serão modificados com valores mínimos (0 ou pimenta). Obviamente, os pixels escolhidos aleatoriamente devem ser posições válidas na imagem de entrada. A função que contamina a imagem com ruído pode tanto ser chamada fora desta função quanto dentro dela.

Um exemplo de resultado para a criação do ruído e da filtragem da mediana aplicado imagens com diferentes níveis de ruído pode ser visto na Figura 23.



Figura 23: Filtragem da mediana para remoção de ruído nas imagens lena.ppm e peppers.ppm.

6.18 Aumento de tamanho da imagem com interpolação bilinear

```
void aumenta_tamanho(Img img_in, Img& img_out);
```

Esta função deve receber como parâmetro de entrada uma imagem e como parâmetro de saída uma outra imagem. Considerando uma imagem de entrada com M colunas e N linhas, o algoritmo deve calcular as entradas de uma matriz com $2M - 1$ colunas e $2N - 1$ linhas correspondente à imagem original em maior resolução (maior tamanho). A função deve aumentar o tamanho da imagem de entrada utilizando o algoritmo de interpolação bilinear, ilustrado na Figura 24.

O processo é realizado em quatro etapas. Na primeira delas, os pixels da imagem original (Figura 24a) são colocados na imagem de saída de forma espaçada, isto é, posicionados alternando linha sim, linha não e coluna sim, coluna não (Figura 24b). Na segunda etapa (Figura 24c), as entradas da matriz da primeira linha, última linha, primeira coluna e última coluna devem ser calculadas pela média aritmética entre os pixels anterior e posterior. Na terceira etapa (Figura 24d), os pixels das linhas e colunas de índice ímpar devem ser calculados como a média aritmética dos pixels vizinhos nas diagonais. Na quarta e última etapa (Figura 24e), os pixels restantes, que possuem soma do índice da linha e coluna não divisível por 2, devem ser calculados como a média aritmética do pixel vizinho de cima, de baixo, da esquerda e da direita. Isto irá resultar na imagem expandida mostrada na Figura 24f. Um exemplo de resultado para o aumento de tamanho de uma imagem com o algoritmo de interpolação bilinear pode ser visto na Figura 25.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(a) Matriz da imagem original.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(b) Etapa 1: cópia dos pixels da imagem original, com entradas a serem computadas pelo algoritmo.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(c) Etapa 2: entradas da primeira e última linha e coluna.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(d) Etapa 3: entradas que dependem de vizinhos diagonais.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(e) Etapa 4: entradas que dependem dos vizinhos esquerdo, direito, de cima e de baixo.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(f) Matriz expandida, após todos os elementos calculados.

Figura 24: Intuição do algoritmo para aumentar o tamanho de uma imagem. As entradas da matriz em verde são iguais às da imagem original, as em cinza são as que restam ser calculadas, as em vermelho são as que estão sendo calculadas em uma dada etapa e as em amarelo são as que foram calculados em uma etapa anterior do algoritmo. Os índices nas entradas verdes se referem à posição de cada pixel na imagem original.

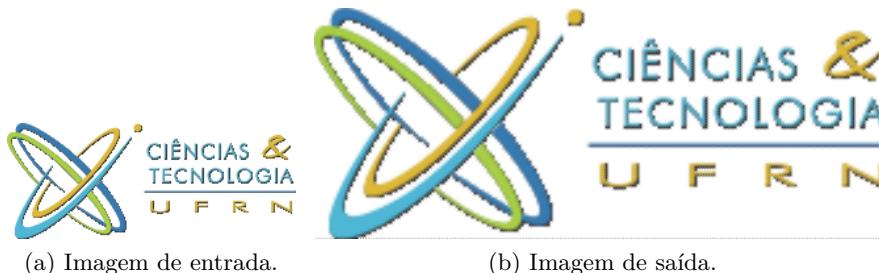


Figura 25: Aumento de tamanho em ect_logo.ppm

Referências

- [1] Rafael C. Gonzalez e Richard E. Woods, *Digital Image Processing*, Prentice Hall Inc., 3ra. edição, 2006.

- [2] Netpbm, *Especificação do formato .ppm*, Online: <http://netpbm.sourceforge.net/doc/ppm.html>, Acessado em 6/11/2016.