MC920 – Trabalho 1

Rafael Sartori M. Santos, 186154

11 de setembro de 2019

1 Introdução

O objetivo do trabalho é analisar, no processo de *half-toning*, os diferentes métodos de difusão de erro: Floyd-Steinberg, Sevenson-Arce, Burkes, Sierra, Stucki e Jarvis-Judice-Ninke. Aplicarei uma transformação em 2 níveis para cada camada de cor de uma imagem representada com 3 camadas: vermelho, verde e azul (R, G, B). Obteremos ao final 8 possibilidades de cor para cada ponto da imagem. Analisarei uma imagem de grande dimensão no formato PNG quanto a sua semelhança à original em uma tela de computador em 2 casos: curta e longa distância de visualização.

Farei esse processamento utilizando Python com as bibliotecas *OpenCV* e *NumPy* em um *Jupyter Notebook*.

2 Método

Capturei uma imagem de grande dimensão (3096x4128) colorida com meu celular sob condições de alta luminosidade, pois gostaria de avaliar o efeito produzido em imagens que são comuns ao dia a dia. Converti do formato original JPEG para PNG utilizando a ferramenta *GIMP*. No tratamento, utilizo apenas a PNG para não possuir intererência da compressão com perdas do outro formato.

Para realizar o processamento digital, as bibliotecas de Python que utilizei foram:

- *OpenCV* para abrir e salvar imagens;
- *NumPy* para aplicar transformações à imagem;
- Alguns módulos da padrão para calcular tempo de execução e iterar de formas diferentes na imagem.

Organizei todo código responsável pelo processamento e medição de tempo em um *Jupyter Notebook*; o que era responsável por abrir e salvar imagens e aplicar as conversões necessárias para utilizar nas bibliotecas em um módulo de utilidade (comum a outros trabalhos da disciplina).

Escolhi manter apenas 2 níveis de intensidade para cada camada de cor da imagem. Como há 3 camadas na imagem testada, obtive 2^3 possibilidades de cores para cada ponto.

2.1 Meios-tons

Começo separando as camadas da imagem colorida de N cores em N matrizes (uma para cada cor). A aplicação de meios-tons na imagem f para produzir a imagem g é dada pela eq. (1).

$$g(x,y) = \begin{cases} 255 & \text{se } f(x,y) \geq 128 \\ 0 & \text{caso contrário} \end{cases} \tag{1}$$

Podemos realizar essa operação de forma vetorial quando nenhuma difusão de erro é aplicada. Caso contrário, precisamos alterar a imagem ponto a ponto e, por conta disso, a vetorização é limitada ao cálculo da difusão.

2.2 Aplicação da difusão de erro

Represento cada método de difusão através de uma matriz, que chamarei simplesmente de filtro, cujo centro de aplicação deve ser mencionado para sabermos onde aplicá-lo de forma vetorial. A difusão de Floyd-Steinberg, por exemplo, é representada pelo par matriz e centro de aplicação:

$$\left(\begin{bmatrix} 0 & 0 & 7/16 \\ 3/16 & 5/16 & 1/16 \end{bmatrix}, (0,1) \right)$$

Para fazermos a aplicação de forma ainda mais efiicente, que é necessário pelo tamanho da imagem, precisamos adicionar *padding* na imagem de entrada. O tamanho da borda é o mínimo necessário (calculado a partir do tamanho do filtro) para que o filtro possa ser aplicado sem condicionais.

A abordagem de *padding* que tomei faz com que o filtro preencha também a borda que é excluída na imagem final, ou seja, os valores dela não são visíveis.

Para percorrer a imagem, criei um *iterator* para a imagem que realiza o zigue-zague na imagem como

recomendado para a atividade: em linhas ímpares, percorremos a imagem crescentemente; nas pares, decrescentemente. Ele retorna em cada ponto a tupla composta pela coordenada horizontal, vertical e ainda o filtro, que é espelhado em relação à vertical nas linhas pares.

O *iterator* é implementado de forma a reduzir os condicionais necessários enquanto navega a imagem através de ponteiro para função que é chamada a cada iteração, atualizada a cada linha. Outro benefício de utilizar essa interface é facilitar a implementação de outros caminhos: basta substituir o *iterador*.

2.3 Limitações

Como, para disseminar erros, é necessário alterar a imagem de entrada na aplicação, não é possível vetorizar toda a aplicação do filtro como é possível realizar em alguns casos. Por conta isso, só consegui vetorizar a aplicação da difusão de erro em cada ponto da imagem e a correção de valores (para manter a imagem no intervalo $[0, 255]$). Isso resulta em um código que demora vários minutos para imagens grandes: para a imagem que utilizei, o tempo de execução foi entre $879,98$ e $918,84$ segundos (aproximadamente 15 minutos).

Também vale ressaltar que tentei executar em paralelo o processamento de cada camada utilizando a biblioteca padrão `multiprocessing`, mas houve problemas de compartilhamento de memória entre processos quando a imagem possui grandes dimensões.

3 Resultados obtidos e análise

Há algumas considerações que valem para todas as imagens. Por exemplo, apesar da abordagem que utilizo para aplicação do filtro alterar a borda, perdendo valores que poderiam ser mantidas na imagem, não foi possível notar algum artefato específico às bordas nas respostas.

3.1 Binarização sem difusão de erro